1999

Optimal power flow using a genetic algorithm and linear algebra

Reid S. Maust
West Virginia University

Optimal Power Flow Using a Genetic Algorithm and Linear Algebra

Reid S. Maust

A DISSERTATION

Submitted to the
College of Engineering and Mineral Resources
at
West Virginia University

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Engineering

Ronald L. Klein, Ph.D., Chair
Muhammad A. Choudhry, Ph.D.
Ian Christie, Ph.D.
Parviz Famouri, Ph.D.
Ali Feliachi, Ph.D.

Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
1999

Keywords: Optimal Power Flow, Genetic Algorithms, Economic Dispatch

Abstract

Optimal Power Flow Using a Genetic Algorithm and Linear Algebra

Reid S. Maust

Artificial intelligence is used to help a hypothetical electric utility meet is electric load economically. The optimal power flow problem (OPF) problem is an optimization problem, in which the utility strives to minimize its costs while satisfying all of its constraints. A genetic algorithm (GA)—a specific type of artificial intelligence—is employed to perform this optimization. To speed convergence, some theory from linear algebra is incorporated into the algorithm.

A GA provides several advantages over more traditional OPF algorithms. For instance, a GA does not constrain the shape of the generators' cost curves and is flexible enough to incorporate control devices such as tap-changing transformers and static VAR compensators.

In the literature, GA-based methods typically use the GA to find the real power and voltage magnitude at each generation bus. To enforce the inequality constraints on voltage magnitudes and angles, these algorithms must compute these quantities for all buses. This requires the solution of the load-flow equations, a set of nonlinear equations that provide real and reactive power in terms of voltage magnitude and angle. Solving for the voltage quantities is computationally intensive when performed repeatedly through the iterations of a method. In contrast, the GA-OPF method presented here reduces the number of load-flow solutions by having the GA find the voltage magnitude and angle at each bus. The real and reactive power are then found by direct substitution into the load-flow equations. To narrow the search for the optimal solution, a vector space is derived that contains all solutions meeting the inequality constraints. This speeds convergence of the algorithm by eliminating a large number of illegal solutions.

The effectiveness of this method is demonstrated on three test systems—the Steinberg and Smith example, the IEEE 30-bus test system, and the IEEE 118-bus test system. For the first two examples, the GA-OPF algorithm finds an answer that agrees with published results. For the 118-bus system, the GA-OPF demonstrates its ability to enforce emission constraints and its potential to be used with larger systems. Thus, the GA-OPF algorithm is shown to be a valid tool to perform this optimization.

Acknowledgments

I would like to acknowledge the contributions of my advisory committee to this work. Dr. Muhammad A. Choudhry, Dr. Ian Christie, Dr. Parviz Famouri, and Dr. Ali Feliachi provided invaluable insights to this work, both inside and outside of their well-taught courses. I wish to further acknowledge Dr. Feliachi specifically for his guidance about the theory of the optimal power flow problem.

Finally, I would like to acknowledge the chair of my advisory committee, Dr. Ronald L. Klein, who funded this research. His interest in genetic algorithms inspired their use here to solve the problem. Dr. Klein's advice throughout the work is greatly appreciated.

Table of Contents

Table of Figures

Chapter 1. Introduction

With the onset of deregulation and competition, electric utilities have new incentives to reduce their costs. Since a major component of operating cost is the cost of the fuel to power the generators' turbines, the electric industry has shown an increasing interest in reducing fuel costs. A method is proposed here to minimize these costs by improving the *optimal power flow* (OPF) algorithm, which is responsible for finding the optimal division of electric load (including transmission losses) among the available generation units. In other words, OPF is an *economic dispatch* (ED) algorithm that accounts for losses.

Given the dependence of each generator's fuel costs on the load it supplies, the objective of the OPF algorithm is to allocate the total electric power demand (and losses) among the available generators in such a manner that minimizes the electric utility's total fuel cost [1–10]. In practice, however, many common economic dispatch algorithms are not flexible enough to allow accurate modeling of the fuel costs. Most common ED algorithms are based on setting the incremental generation costs (essentially incremental fuel costs) of each generator equal to one another, perhaps with some adjustment to account for losses [1–4]. For the equal-incremental-cost solution to be optimal, each generator's incremental cost curve must be a monotonically increasing function of load, which is not necessarily the case for a physical generator [2,4,8].

Complicating matters is the fact that OPF is a constrained optimization. The load-flow equations are equality constraints on the solution, while limits on quantities such as power generation, voltage magnitude, and line flows are inequality constraints. Thus, analytic solution requires the use of such techniques as Lagrange multipliers and the Kuhn-Tucker method to enforce these constraints [1,2]. Some researchers, such as Bakirtzis [6], linearize the problem and employ linear programming to perform the optimization. Recently, in an effort to avoid the difficulties of enforcing constraints, techniques employing artificial intelligence to ED, OPF, and related problems have begun to appear in the literature [5–8]. In this project, a *genetic algorithm (GA)*, a specific type of artificial intelligence, is used in a new way to solve the OPF problem.

This work makes the following contributions to the application of GA to OPF:

1. The definition of a new genetic chromosome structure to represent the solutions. The new chromosome structure is chosen in such a way that it greatly reduces the number of times the algorithm must solve the load-flow equations. Since solving the load-flow equations is time-consuming, this speeds execution of the algorithm considerably.

2. The use of linear algebra's nullspace theory to reduce the search space, which prevents the algorithm from spending a great deal of time evaluating illegal solutions.

3. The derivation of equations to represent changes in transformer tap settings in a way consistent with the nullspace representation.

To demonstrate the effectiveness of the GA-OPF method, it is tested on three test systems of varying complexity. For one test system, the GA-OPF method is altered to demonstrate the enforcement of emission constraints.

Chapter 2. Literature Review

In this section, a brief description is given of some relevant previous work. First, optimal power flow, the problem being solved, is described. Second, an overview of genetic algorithms (a form of artificial intelligence) is given. A genetic algorithm is used here to solve the optimal power problem. Third, a technique from linear algebra is presented as an analytical tool that narrows the possibilities that must be considered by the algorithm. Fourth, three test systems are defined, to allow quantitative evaluation of the algorithm. Fifth, to address increasingly stringent environmental requirements, a method is presented to allow the modeling of emissions constraints.

2.1 Optimal Power Flow

Given each generator's cost to generate a given amount of electric power, a utility must determine the optimal amount of power to be supplied by each generator. This optimization is divided into three problems, which differ in their time horizon [1]. Looking ahead a day or two is the *unit commitment problem*, in which a typical utility uses forecasts for the next day's power demand to decide which generators to bring on-line. Looking ahead a few minutes is the *economic dispatch* problem, in which the utility decides how much power should be supplied by each generator. In real time (or nearly in real time) *automatic generation control* is performed to correct any mismatch between power generated and used. This work will investigate the optimal power flow problem, which is economic dispatch while accounting for transmission losses. Some of the methods of solving the unit commitment problem [7] are adapted here for the OPF problem.

2.1.1 Equations for the Optimal Power Flow Problem

In order to compute the power flows in a power system, the system's bus admittance matrix, Y_{BUS}, must be defined. If V and I are respectively vectors of all voltages and currents in the system, the bus admittance matrix will satisfy [2]

$$I = Y_{BUS} V \qquad \textbf{(1)}$$

where Y_{BUS} is a square matrix which depends on the admittance of all transmission lines in the system. Let y_{Si} be the shunt admittance connected at bus i, and let y_{ij} be the series admittance connecting buses i and j. Note that y_{ij} equals zero if buses i and j are not connected. The elements of Y_{BUS} are defined as [2]

$$(Y_{BUS})_{ij} = \begin{cases} -y_{ij} & i \neq j \\ y_{Si} + \sum_{m \neq i} y_{im} & i = j \end{cases} \qquad \textbf{(2)}$$

In the optimal power flow problem, it is necessary to find a relationship between the voltage magnitudes and angles and the real and reactive power at the buses. For bus l, let V_l and d_l be the voltage magnitude and angle, respectively. Furthermore, let the P_{Gl} be the real power generated, let P_{Dl} be the real power demand (the real power load), let Q_{Gl} be the reactive power generated, and let Q_{Dl} be the reactive power demand. Then, the net real and reactive power at bus l are given by the load-flow equations [1]:

$$P_l = P_{Gl} - P_{Dl} = V_l^2 G_{ll} - V_l \sum_{m \in k(l)} V_m T_{lm} \qquad \textbf{(3)}$$

$$Q_l = Q_{Gl} - Q_{Dl} = -V_l^2 B_{ll} - V_l \sum_{m \in k(l)} V_m U_{lm} \qquad \textbf{(4)}$$

where

$$T_{ij} = G_{ij}\cos(d_i - d_j) + B_{ij}\sin(d_i - d_j) \quad \textbf{(5)}$$

$$U_{ij} = G_{ij}\sin(d_i - d_j) - B_{ij}\cos(d_i - d_j) \quad \textbf{(6)}$$

and where G_{ij} and B_{ij} are respectively the real and imaginary parts of the (i,j) element of Y_{BUS}.

The OPF problem also defines a Jacobian matrix, which is a matrix of partial derivatives of power quantities with respect to voltage magnitude and angle. The system Jacobian matrix is partitioned into four submatrices, each of which is an $N \acute{} N$ matrix [1]:

$$J = \begin{bmatrix} \frac{\partial P}{\partial d} & \frac{\partial P}{\partial V} \\ \frac{\partial Q}{\partial d} & \frac{\partial Q}{\partial V} \end{bmatrix} \quad \textbf{(7)}$$

Let k(i) be the set of all buses connected to bus *i*. In defining the submatrices, let the indices *i* and *k* be row and column positions within each submatrix. Then, the elements of the Jacobian's submatrices are [1]

$$\frac{\partial P_i}{\partial d_i} = V_i \sum_{j \in k(i)} V_j U_{ij} \quad \textbf{(8)}$$

$$\frac{\partial P_i}{\partial d_j} = -V_i V_j U_{ij} \quad \textbf{(9)}$$

$$\frac{\partial P_i}{\partial V_i} = 2V_i G_{ii} - \sum_{j \in k(i)} V_j T_{ij} \quad \textbf{(10)}$$

$$\frac{\partial P_i}{\partial V_j} = V_i T_{ij} \quad \textbf{(11)}$$

$$\frac{\partial Q_i}{\partial d_i} = -V_i \sum_{j \in k(i)} V_j T_{ij} \quad \textbf{(12)}$$

$$\frac{\partial Q_i}{\partial d_j} = V_i V_j T_{ij} \quad \textbf{(13)}$$

$$\frac{\partial Q_i}{\partial V_i} = -2V_i B_{ii} - \sum_{j \in k(i)} V_j U_{ij} \quad \textbf{(14)}$$

$$\frac{\partial Q_i}{\partial V_j} = -V_i U_{ij} \quad \textbf{(15)}$$

Note that the Jacobian is defined in terms of T_{ij} and U_{ij}, which are themselves defined in terms of the elements of Y_{bus}. Since the Jacobian is partitioned into a 2×2 array of submatrices that all depend on T_{ij} and U_{ij}, changing one element of Y_{bus} could conceivably affect several elements of J. Upon inspection of the above expressions for elements of the Jacobian, note that each transformer's 2×2 submatrix of Y_{bus} in turn affects a 4×4 submatrix in the Jacobian—a 2×2 submatrix in each of the Jacobian's four partitions. Again, let P and S refer to the bus numbers of the transformer's primary and secondary windings. Let N be the total number of buses in the system. Thus, the Jacobian is a matrix of size $2N \acute{} 2N$.

2.1.2 Adjusting Y_{BUS} for Changes in Transformer Taps

In the traditional OPF and ED strategies, accounting for changes in transformer tap value is straightforward. Every time a transformer's tap value is changed, a new set of parameters is found for the equivalent-pi circuit, whose schematic is shown in Figure 1.

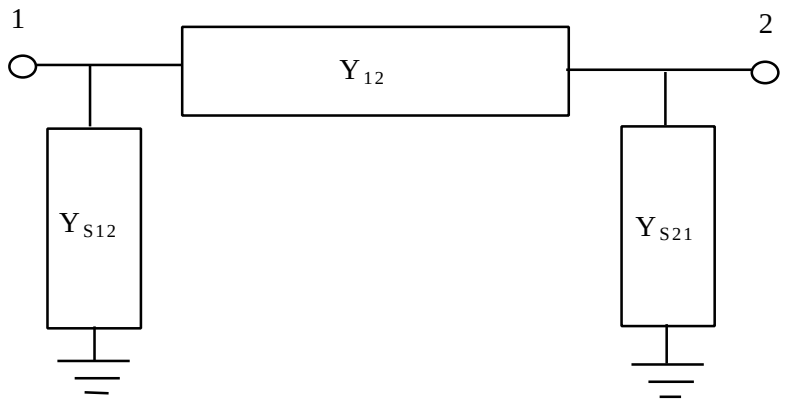

Figure 1. General equivalent-pi circuit

The equivalent-pi admittances are defined in terms of the transformer's admittance (or impedance) and turns ratio [1,2]. The new equivalent-pi admittances are then incorporated into the system's bus-admittance matrix (Y_{bus}). Following Debs' notation, let Y_L equal the transformer's series admittance and let t equal its turns ratio. Then, the equivalent-pi parameters in Figure 1 are [1]

$$Y_{12} = tY_L \quad \textbf{(16)}$$

$$Y_{S12} = -t(1-t)Y_L \quad \textbf{(17)}$$

$$Y_{S21} = (1-t)Y_L \quad \textbf{(18)}$$

Of course, it is always possible to create a new Y_{bus} matrix from scratch whenever the taps are changed. However, Gross [2] simplifies the equations and derives the changes to Y_{bus} caused by changes in taps. He notes that any given transformer's tap setting affects only four elements of Y_{bus}—the 2×2 submatrix formed by the intersection of the rows and columns corresponding to the primary and secondary buses. However, caution is required when applying Gross' equations to Debs' model. Debs assumes that the transformers have a turns ratio of 1:t, while Gross assumes a ratio of c:1. Thus, to use Gross' equations, it is necessary to note that c = 1/t. With this substitution, Gross' equations (converted to Debs' notation) become [2]

$$\Delta Y_{12} = (t_0 - t)Y_L \quad \textbf{(19)}$$

$$\Delta Y_{PP} = (t^2 - t_0^2)Y_L \quad \textbf{(20)}$$

$$\Delta Y_{BUS,2\times 2} = \begin{bmatrix} \Delta Y_{PP} & \Delta Y_{12} \\ \Delta Y_{12} & 0 \end{bmatrix} \quad \textbf{(21)}$$

where Y_L is the series admittance of the transformer, and a subscript of *0* corresponds to the original values. The new values are written with no subscript. The variables *P* and *S* refer to the bus numbers of the primary and secondary windings of the transformer, respectively. The notation $Y_{BUS,2'2}$ refers to the transformer's 2×2 submatrix—2 rows and 2 columns. The first row and column correspond to the primary; the second row and column correspond to the secondary. Note that $Y_{BUS}(S,S)$, the diagonal element corresponding to the secondary, does not change.

Gross' equations are applied to one transformer at a time. If more than one transformer has changed its taps, superposition is used; the individual effects of each transformer are summed to find the aggregate change in Y_{bus}.

2.1.3 Optimization Performed by Economic Dispatch and Optimal Power Flow

Traditional economic dispatch methods are based on setting incremental costs of all units equal to each other [1,2,8]. Losses are accounted for by incorporating penalty factors in the incremental cost [2,8]. However, the equal-incremental-cost method is optimal only if the incremental cost curves are monotonically increasing [4,8], which is not always true. In fact, for certain cost curves, the equal-incremental-cost solution has the *highest* possible fuel cost [4]. In practical applications, the incremental cost functions are often constrained to be monotonically increasing, regardless of the generator's actual behavior [2]. This is done to allow the use of standard economic dispatch algorithms, even at the expense of accuracy [2].

Steinberg and Smith's example [4] illustrates the inadequacy of traditional economic dispatch when the incremental cost curves are not monotonically increasing. This provides the motivation for applying a genetic algorithm to the OPF problem. To

illustrate the method presented here, it is performed on Steinberg and Smith's example [4], which is described in Section 2.4.1.

2.1.4 Load-flow Solution

Given each generator's real power and voltage magnitude as well as the system load, a *load-flow solution* is the solution of a set of nonlinear equations to find voltage magnitude and angle at load buses, reactive power at generation buses [1,2]. This is necessary when checking if voltages violate their constraints. A common method for solving the load-flow equations is the Newton-Raphson method [1,2]. However, the Newton-Raphson method has the disadvantage of requiring each iteration to re-evaluate and invert a Jacobian matrix [2]. For a realistically sized power system, the Jacobian is a large matrix, and the inversion is time-consuming. An alternative method is the *Fast Decoupled Load-Flow* (FDLF) solution, which partitions the Jacobian into a 2×2 collection of subarrays and then neglects the off-diagonal subarrays [1,2,11]. This reduces the load flow equations into two simpler, decopuled sets. Two key advantages of the FDLF are [1]

1. The Jacobian is replaced by two constant matrices, which only have to be inverted once, rather than at each iteration.

2. The FDLF has a wider region of convergence than the Newton-Raphson method.

Although the FDLF must perform more iterations than the Newton-Raphson method, the FDLF iterations are much faster than the Newton-Raphson iterations. The FDLF requires about one third as much solution time as Newton-Raphson [1].

2.1.5 OPF Strategies in the Literature

Because the analytic techniques for OPF are well known (albeit difficult to implement), this section will concentrate on iterative OPF or ED methods using three very different strategies: differentiating the performance index, using linear programming, and running a genetic algorithm.

Representing the first strategy, Lee, Park, and Ortiz decompose the OPF problem into two separate modules, one for real power and one for reactive power [5]. This method uses the *gradient projection method* to converge iteratively to a solution. In essence, the system's Jacobian matrix is used to update the control variables (in a method similar to Newton-Raphson). Lee, Park, and Ortiz illustrate their technique on the IEEE 6-bus and 30-bus systems, for which they provide the line impedance data and generator cost data.

Representing the second strategy, Bakirtzis [6] solves the OPF problem for the IEEE 30-bus system problem by iteratively using linear programming. This method converges rapidly, but it requires repeated linearizations of the problem, including the performance index. In order for a global solution to exist, Bakirtzis assumes that the optimization problem is convex [6], which means (in part) that the generators' incremental-cost curves are all monotonically increasing. In contrast, the method presented here avoids both the linearization of the performance index and the constraint on the incremental curves' convexity.

In two variants of the third strategy, Wong and Wong [7] and Bakirtzis et al. [8] use a genetic algorithm to solve an ED problem. Wong and Wong [7] solve a busbar ED problem, which is ED that ignores losses and line-flow constraints. Unlike traditional methods, however, they do not constrain the generators' incremental cost curves. Instead, they use curves that are not smooth but represent the effect of pressure changes as a generator's steam valve is gradually opened [7]. A fully open valve is more efficient than one that is just barely open. Wong and Wong demonstrate the flexibility of a genetic algorithm to solve a problem similar to the OPF problem considered here. Bakirtzis et al. [8] include losses in their solution, through the method of "B-coefficients," which are linearized sensitivity coefficients, representing the effect of power supplied by each generation unit on total system losses.

2.2 Genetic Algorithms

A genetic algorithm (GA) [12,13,14,15] is an optimization technique using artificial intelligence. The method is based on Darwin's survival of the fittest hypothesis. In a GA, candidate solutions to a problem are analogous to individual animals in a population.

Although the initial population can be a random collection of bizarre individuals, the individuals will interact and breed to form future generations. Stronger individuals will reproduce more often than will weaker individuals. Presumably, the population will get collectively stronger as generations pass and weaker individuals die out. The quantitative application of these basic ideas to an actual algorithm is a combination of science and art.

2.2.1 Implementation of a Genetic Algorithm

In a genetic optimization problem, the objective is to maximize a *fitness function*. The fitness is calculated for each member of the population, and some individuals are selected to survive into the next generation. Under *roulette-wheel* selection [12,13], an individual's probability of survival is directly proportional to its fitness value. The selection operation forms the next generation of solutions by copying randomly chosen survivors from the previous generation. It is possible that some very fit functions might be copied into the next generation more than once (cloning), while some unfit functions might not be copied at all (death). Because of the probabilistic nature of this selection mechanism, it is also possible for the best solution to be passed over and not be chosen for survival. This work uses *elitism* [12,13] to guarantee that the best solution will always survive. Once the new generation of solutions is formed, the genetic *crossover* operators form new solutions by combining old solutions according to a predetermined set of rules. Furthermore, genetic mutation operators randomly alter some of the new solutions, in order to add diversity to the population. The choice of crossover operator depends on the problem being optimized and the structure of the solutions.

Because of the manner in which genetic methods use the fitness function, great flexibility is afforded the designer. Unlike other optimization methods, the genetic methods do not impose constraints on the form of the fitness function. Since a GA does not differentiate the fitness function, the fitness function does not need to be differentiable or even continuous. Furthermore, this flexibility allows the direct enforcement of constraints. The GA can be constructed so that it never generates an illegal set of control variables. However, it is still possible that one or more of the dependent (output) variables violates a constraint. If this happens, the designer is afforded the choice of discarding the solution, keeping the solution but penalizing its

fitness value, or repairing the solution in a manner which will make it better fit the constraints [13]. Each of these methods has its individual advantages and limitations, which require analytical and intuitive skills by the designer to select and apply intelligently. The best choice depends on the problem being solved. Discarding illegal solutions guarantees that illegal solutions will not be accepted, but it causes the population to lose diversity. Keeping an illegal solution while penalizing its fitness will allow its survival, thereby keeping its diversity in the population but will not guarantee that the final solution is legal. Repair algorithms require special skill to design and usually slow the execution rate of the algorithm. In this project, illegal solutions will be allowed to survive, but will be penalized.

2.2.2 Strengths and Limitations of a Genetic Algorithm

Genetic techniques have the following advantages over conventional optimization techniques:

1. Because of its iterative nature, a GA can optimize with respect to a nonlinear, analytically intractable performance index.
2. Genetic techniques do not require a differentiable performance index. Thus, this research is not restricted to using the least-square error criterion.
3. GAs can readily enforce constraints on the control variables. In contrast, enforcing constraints using conventional techniques can result in an intractable set of partial differential equations (such as those resulting from setting partial derivatives of the Lagrangian equal to zero).
4. The structure of the optimization technique can become more or less complicated to match the complexity of the problem. There are many variants on the GA method.

These advantages give genetic techniques great flexibility in solving the system identification problem. However, like any computation technique, a GA has limitations. Two important limitations of GAs (and how to lessen their impact) are:

1. Execution time. GAs can require evaluation of thousands of candidate solutions before converging on the best solution. This is a problem for the OPF algorithm. Performing as much of the optimization as possible offline lessens this problem. These coarse optimization results would greatly reduce the search space for the final optimization, which would then fine-tune the results.
2. It is not always obvious that a GA has found the best answer possible. Although genetic techniques are less susceptible to getting trapped in a local (rather than global) optimum than other techniques such as hill climbing or simulated annealing [8], converging to a suboptimal solution is still possible. Increasing the population size, evolving the population for more generations, or increasing the amount of mutation in the population can counteract suboptimal convergence.

Note that GAs are not generally used for problems easily optimized using conventional techniques. For difficult optimization problems, however, the power and flexibility of the genetic techniques outweigh the limitations.

2.3 Use of Linear Algebra to Improve Convergence of the GA

Although a GA is an efficient search technique for large problems [12], its convergence can be improved significantly by encoding the candidate solutions in such a way that avoids generating illegal candidate solutions [13]. For example, equality constraints are difficult to implement with a GA. One technique is to use the equality constraints to solve for some of the control variables in terms of the others [12]. This has the effect of narrowing the search space and reducing the dimensionality of the problem (since there are fewer unknowns remaining). Furthermore, this avoids wasting computation effort unnecessarily on illegal solutions.

In the OPF problem, it is not feasible to use the load-flow equality constraints to eliminate state variables. Enforcing the equality constraints requires solving the load-flow equations, which is a computationally intense task. Instead, the search space is reduced via the representation of the candidate solutions. For a power system with N buses and N_g generation buses, there are $2N$ state variables (voltage magnitude and angle at each bus) but only $2N_g$ control variables (real and reactive power at each generator). If

the GA produced a candidate solution by randomly choosing a list of *2N* state variables, the solution likely would fail to meet the equality constraints. In other words, such a solution is unlikely to have the correct amount of real and reactive power at all $(N–N_g)$ load buses.

Thus, the equality constraints restrict the choice of values for the state variables. Let *J* be the load-flow Jacobian matrix. Here, all buses—even the slack bus—are represented in the Jacobian. Thus, *J* is a $2N \times 2N$ matrix. Let J_L be rectangular matrix formed by taking the rows of *J* corresponding to the load buses. In other words, any partial derivative involving *P* or *Q* at a load bus is kept. Thus, J_L is a $2(N–N_g) \times 2N$ rectangular submatrix of *J*. The matrix J_L has 2 rows for each load bus (corresponding to one *P* and one *Q* for each load bus) and 2 columns for each bus of any kind (corresponding to one voltage magnitude and one voltage angle for each bus, whether it is a load bus or not).

Let *x* be a state vector that satisfies the equality constraints. Any change to the state vector, *Dx*, will change the power vector by

$$\Delta S = J\Delta x \qquad \textbf{(22)}$$

where the state vector, *x*, and power vector, *S*, are defined as

$$x = \begin{bmatrix} d \\ V \end{bmatrix} \qquad \textbf{(23)}$$

$$S = \begin{bmatrix} P \\ Q \end{bmatrix} \qquad \textbf{(24)}$$

and where *d*, *V*, *P*, and *Q* are all N×1 vectors listing the voltage angle, voltage magnitude, real power, and reactive power respectively. But, *P* and *Q* are specified at all load buses. Therefore, to avoid changing the power at these buses, *DS* must contain a *0* in all rows representing a load bus

The load-bus rows of Equation (22) can be extracted to yield

$$\Delta S_L = J_L \Delta x = 0 \qquad \textbf{(25)}$$

where the subscript *L* signifies that only the load-bus rows are retained. The right side of the equation is a zero vector of size $2(N–N_g)\times 1$, which contains two entries (one *DP* and one *DQ*) for each load bus. Thus, Equation (25) (which is derived from the equality constraints) forces *Dx* to lie in the *nullspace* [16] of the rectangular matrix J_L. To define the nullspace of a matrix, let a vector *v* and a matrix *A* are defined so that they satisfy the matrix equation

$$Av = 0 \qquad \textbf{(26)}$$

In this example, *v* is said to lie in the nullspace of *A*. If *A* is invertible, the nullspace will consist of only the trivial solution—a zero vector. However, if *A* is not invertible (for example, if it is not square), then Equation (26) may have nontrivial solutions. The set of all solutions to (26) is defined as the nullspace of the matrix *A*.

In this case, the nullspace will have dimension $2N_g$, which means that exactly $2N_g$ independent parameters are required to specify a particular solution to (25). Thus, instead of choosing *N* state variables, the algorithm represents each candidate solution by a list of $2N_g$ coefficients, which specifies one vector in the nullspace. Since a GA works with a population of candidate solutions, one set of coefficients is required for each member of the population.

The preceding discussion has neglected the effects of compensation devices such as tap-changing transformers or static-VAR compensation (capacitor banks). The primary effect of these devices is to attempt to keep the voltage at each bus within its allowable range. These devices alter the reactive power (and also the real power, to a much lesser extent) at the buses they connect, which has the effect of changing the load flow solution (the state vector *x* corresponding to the new power vector *S*). Thus, in the presence of these devices, the change in the state vector, *Dx*, has two components: the nullspace component described earlier and a new component to account for the change in power caused by these devices.

2.4 Test Systems

In order to illustrate the effectiveness of the method presented here, it is demonstrated on three test systems: Steinberg and Smith's example, the IEEE 30-bus system, and the IEE 118-bus system. The systems are listed in order of increasing complexity, to illustrate the evolution of the algorithm.

2.4.1 Steinberg and Smith's example

Steinberg and Smith [4] illustrate how the cost curves' curvature affects the optimization. In their example, two isolated machines are supplying a load. The term "isolated" means that the machines are not connected to any other power system. Losses are neglected. Figure 2 shows the heat-input curves for the machines. Heat input is defined as the amount of heat (such as burning coal) required to generate a given amount of electric power. In this example, each machine's output must be between 5 and 80 MWh. For the purposes of this example, generation cost is assumed to be proportional to heat input. Thus, the heat-input curve can be regarded as a generation cost curve.

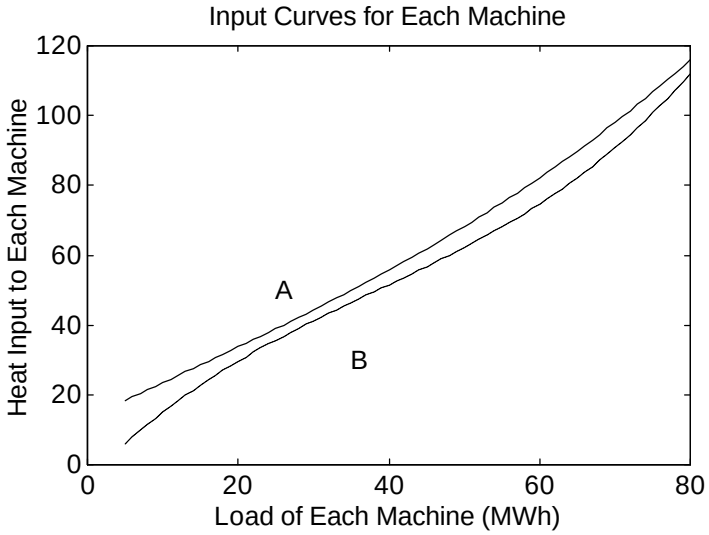

Figure 2. Steinberg and Smith's sample input curves

In their example, Steinberg and Smith provide graphs of their functions but do not give the mathematical expressions for their functions. Therefore, the graphs in this example were found by curve fitting to produce a graph matching the corresponding

graph in [4]. The heat-input curve was assumed to be a cubic function in electric output, and closely resembles the original graph.

The derivative of the heat-input curve is the incremental heat-rate curve, which is given in Figure 3. This curve can be regarded as the incremental cost of generation for each machine, which is the cost of producing the next unit of electricity.

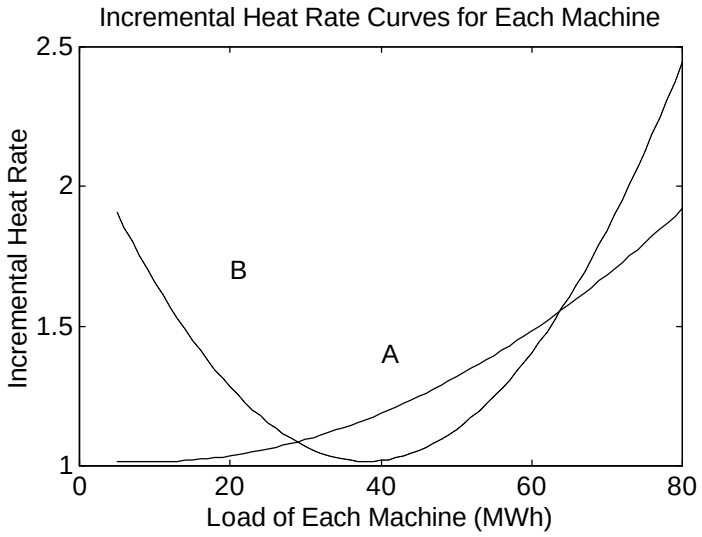

Figure 3. Steinberg and Smith's incremental heat-rate curves

Note that machine B's incremental heat-rate curve is not monotonic. Because of this, standard economic dispatch algorithms will not provide the optimal solution. Since this is a busbar economic dispatch example with only two machines, the problem is greatly simplified. It is unnecessary to account for line losses, and voltage constraints are ignored. Therefore, a simplified genetic algorithm is used to distribute the real-power load between the machines. This simple example illustrates the power of a genetic algorithm to optimize a system without monotonic incremental costs.

2.4.2 IEEE 30-bus system

Although the simplified GA showed promise with Steinberg and Smith's example, the method presented here is performed on the IEEE 30-bus system, in order to provide a more complex test of the algorithm. The generation cost obtained in this method presented here is compared with Alsac and Stott's result [10]. The system has 6

generators and static VAR compensation available at two specified buses. The line impedance values and cost data are given in [10]. The line limits (maximum power flow permitted on the lines) are given in [5]. The data for this system are given in Appendix A.

2.4.3 IEEE 118-bus system

In order to demonstrate the method's performance with a larger system, the algorithm is used with the IEEE 118-bus system. Since a complete, comprehensive model is not readily available, the data for the 118-bus system have been gathered from a variety of sources. The line impedance data comes from the University of Washington archive [17]. The number of generators, location of generators, and location of VAR compensation, generator cost data, and limits on real and reactive power are found in Reid and Hasdorff [18]. For the purposes of this work, all voltage magnitudes are constrained to be between 0.90 and 1.10 p.u., which is the range containing the voltages in [18].

The 118-bus system data are given in Appendix B.

2.5 Modeling of Emissions

With growing political concerns about the environment, it is desirable to adjust the dispatch algorithm to account for emissions. As quoted by the IEEE PES Power System Engineering Committee [19], Southern California Edison has used curve fitting to derive a quantitative model for NO_X. In this work, NO_X is used as a proxy for all emissions. Including other types of emission would be an analogous procedure. The Southern California Edison model is a set of parametric curves, as shown in Figure 4.

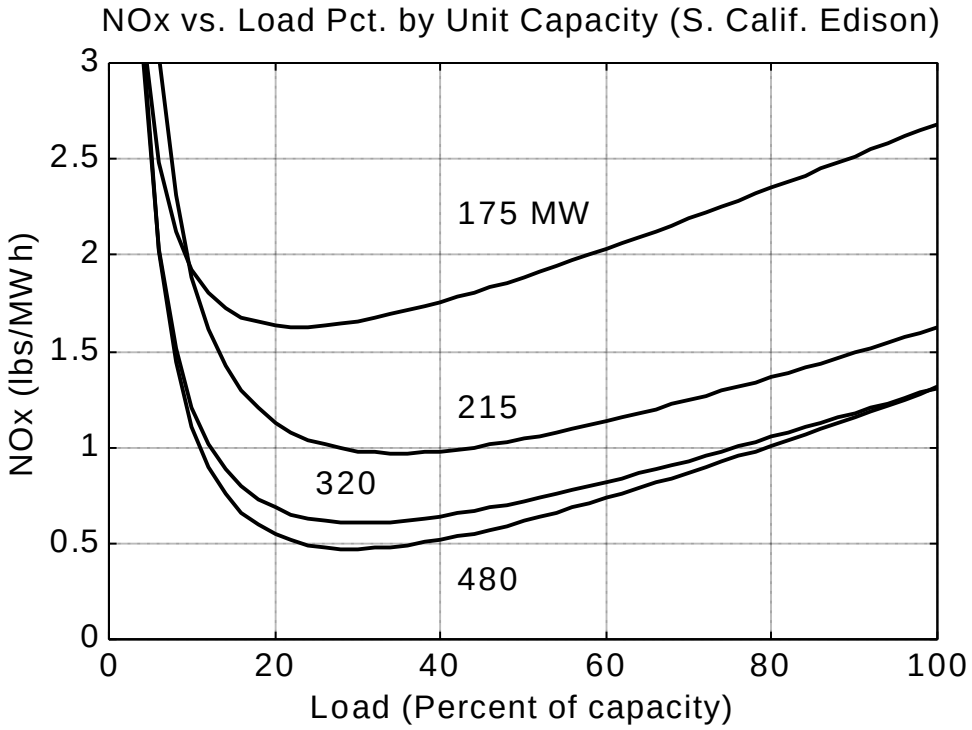

Figure 4. NOx as a function of Unit loading percentage, with unit capacity as a parameter

At first glance at Figure 4, it may appear that the units all produce comparable amounts of NO_X (except possibly for 175-MW units). Note, however, that the larger units generally produce less NO_X for the same *percentage* load. For example, a 480-MW unit produces 0.5 lb NO_X/MWh when it is loaded at 192 MW (40% of 480) wile a 215-MW unit produces 1.0 lb NO_X/MWh when it is loaded at 86 MW (40% of 215).

To better illustrate the difference in NO_X performance of the units, the curves in Figure 4 are altered to plot actual NO_X in pounds per hour vs. actual load, rather than plotting NO_X in lb/MWh vs. percentage load. The derived curves are shown in Figure 5, which does not appear in [19]:

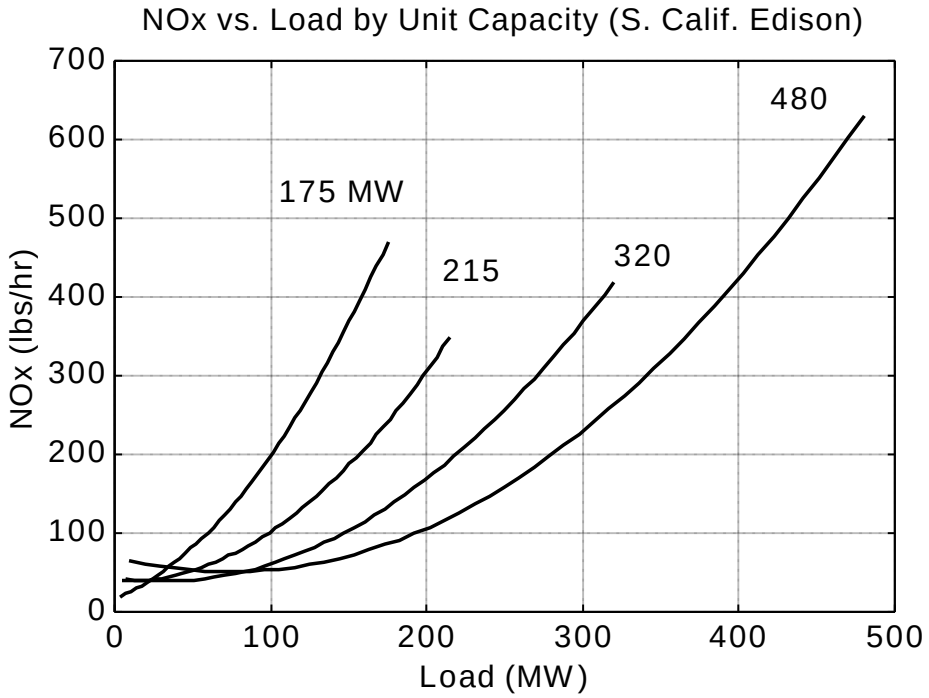

Figure 5. NOx as a function of unit load, with unit capacity as a parameter

Figure 5 demonstrates that the larger units generate less NO_X than smaller units at the same *absolute* load (in MW) and that the difference is significant. For example, to generate 200 MW, the 215-MW unit would produce more than three times as much NO_X as the 480-MW unit (approximately 300 lb/hr of NO_X for the 215-MW unit vs. 100 lb/hr of NO_X for the 480-MW unit). Thus, switching load from one unit to another can have a large impact on emissions.

Chapter 3. Problem Statement

The GA-OPF method will be demonstrated on the IEEE 30-bus [5,10] and 118-bus [17,18] test systems.

Given the data for the chosen test system, minimize the total generation cost which is often modeled as

$$C_T = \sum_{i=1}^{N} (a_i + b_i P_{Gi} + c_i P_{Gi}^2) \qquad \textbf{(27)}$$

where N is the number of generators, and P_{Gi} is the power generated by the i^{th} generator. For the IEEE 30-bus and 118-bus systems examined here, the cost function of each generator happens to be quadratic (as shown in the equation above). However, the algorithm presented here does not require this to be the case. Steinberg and Smith's example has cubic cost curves. In fact, the cost curves are not required even to be continuous.

This optimization is subject to

1. The load-flow equality constraints

2. The inequality constraints: limits on generated real and reactive power, phase angle (absolute value less than 90°), VAR compensation, transformer tap settings, and on line flows. For the 118-bus case, emission constraints are also enforced.

3. The fact that the transformer tap settings and static-VAR compensation are discrete quantities. Note that there are new power electronic devices that allow the tap-settings to be analog variables. However, to be compatible with older equipment that may still be in use, the discrete-quantity assumption is retained for the purposes of this proposal. Here, transformer tap settings and static-VAR compensation are both assumed to be discretized in increments of 0.01.

Chapter 4. Solution

The solution is composed of four parts: selecting the control variables, choosing the genetic operators and fitness function, customizing the GA for the problem at hand, and applying the load-flow equations efficiently.

4.1 Choosing the Control Variables

In the OPF problem, there are four important quantities: voltage magnitude, voltage angle, real power, and reactive power. Of these four quantities, two are independent (control, or input) variables and two are dependent (output) variables. For a traditional OPF problem, the unit incremental cost functions are used to optimize the real and reactive power (which are the control variables in this formulation). Mathematically, the choice of independent variables is not important. For computational speed, however, choosing voltage magnitudes and angles as the independent variables will allow the algorithm to avoid solving load-flow problems for each candidate solution. Although one load-flow problem may not require a great deal of speed, evaluating many load-flows (one for each member of the population, at each generation) is quite slow.

GA convergence is much improved if redundant control variables are removed, and only an independent subset is considered. That is, it is often beneficial to use the equality constraints to eliminate unnecessary control variables [12]. Moreover, to reduce computational effort spent on illegal solutions, the linear algebra nullspace technique is used to reduce the search space. The nullspace eliminates many (but not all) illegal solutions before they are considered. Thus, for this OPF problem, the GA control variables are chosen as:

1. Nullspace coefficients, to specify which member of the nullspace is used
2. Tap settings for the tap-changing transformers
3. Amount of VAR compensation

Each GA chromosome is a list of numbers that provides the values of these control variables. To change the transformer tap settings, the system Y-bus matrix is modified to account for the transformer's new impedance.

Once all control and output variables are known, the fitness of the candidate solution is computed.

4.2 Choosing the Genetic Operators and Fitness Function

A genetic *operator* is a set of rules for extracting new solutions from older ones. The selection of genetic operators is often a heuristic process. A *fitness* function is defined to quantify the quality of any particular candidate solution. A good choice of operators and fitness function for one type of problem can be a poor choice for another problem. Sometimes, the choice of operators depends on the choice of fitness function. Thus, the fitness function has been included in this discussion of genetic operators.

4.2.1 Fitness Function

For this project, the fitness function was chosen to be similar to that of Wayer [15]:

$$f = \frac{1}{1 + C_T + P} \qquad \textbf{(28)}$$

where C_T is the total generation cost and P is the penalty if any output variable violates a constraint. This penalty is the weighted sum, over all output variables, of the amount each variable exceeds its constraint. Of course, if a variable is within its allowable limits, its contribution to the penalty is zero. The weighting factors are chosen to be 10,000 for voltage magnitudes, 10,000 for line flows, and 1000 for all other variables. This choice of fitness function maps a cost in the interval $[0,\infty)$ to the interval $(0,1]$. Thus, a solution with an infinite cost (or infinite penalty) has a fitness of 0. A perfect solution (one with zero cost) has a fitness of 1.

Note that this penalty weight is not the price of power or of anything else. Instead, the weight is a coefficient set large enough to prevent the algorithm from converging to an illegal solution.

Care must be taken not to choose an excessively large penalty weight. If the weight is too large, an illegal solution (even one that is almost within its limits) will have a fitness close to 0. For excessive penalty weights, any perturbation of the illegal solution would also have a fitness close to 0, and thus the fitness values do not give the GA any indication of the best way to improve the solution. Instead, the GA would wander around aimlessly and perform poorly [12]. The weight must be small enough to allow the algorithm to improve an illegal solution (and hopefully make it legal), but the weight must be large enough so that the algorithm does not ignore the constraints. If the weight is too small, the GA will simply pay the penalty for being illegal and not bother to force the solutions to be legal.

4.2.2 Genetic Operators

Crossover operators are used to generate new solutions by taking information from previous solutions. Since the GA used here works with lists of real numbers, two crossover operators used here are arithmetic crossover [12] and two-point crossover [12]. These operators have the advantage that they will always generate a set of control variables within their allowable ranges, provided that the original solutions were legal. However, these operators do not guarantee that a solution will satisfy the other constraints (such as line-flow limits), even if the parents satisfied them.

To illustrate arithmetic crossover, let x_1 and x_2 be vectors containing the coefficients of two "parents"—candidate solutions chosen to participate in the crossover. The two "children"—new candidate solutions resulting from the crossover—are formed by taking two weighted averages of the parents. Let a be a random number between 0 and 1. Arithmetic crossover calculates the children according to the following equations [12]:

$$y_1 = ax_1 + (1-a)x_2 \quad \textbf{(29)}$$

$$y_2 = (1-a)x_1 + ax_2 \quad \textbf{(30)}$$

In contrast, two-point crossover combines information from two parents in a fundamentally different way. It literally breaks the parents apart, exchanges some of the pieces, and recombines the pieces to form two new solutions. This is illustrated in Figure

6, which shows one example of how the operator might produce children from two arbitrary parents.

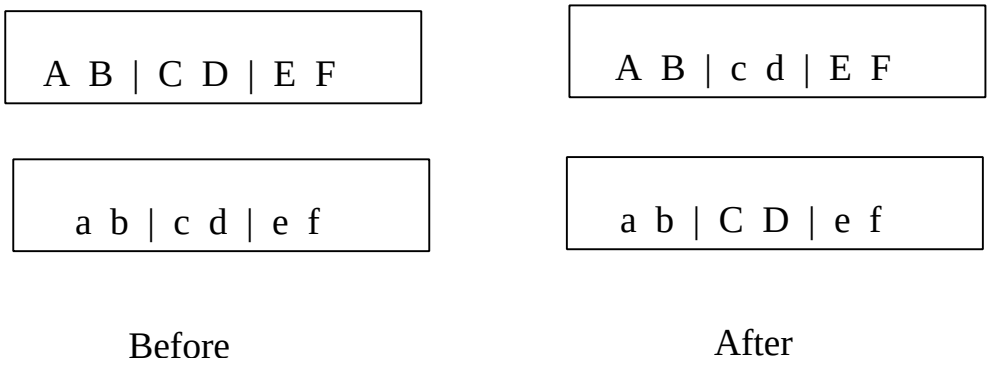

Figure 6. Illustration of Two-point Crossover

For illustrative purposes, the chromosomes (the subdivisions of the parents) are represented by the letters *A–F* and *a–f*. In the OPF problem, the chromosomes are real numbers. The crossover operator randomly selects the portion of the parents it will alter. In this example, it is assumed that the operator will cut the parents at the positions indicated by the vertical bars—after the second and fourth positions. The two vertical bars indicate the "two points" which give this operator its name. The effect of two-point crossover is to exchange all chromosomes appearing between the two points.

Mutation operators are used both to avoid premature convergence of the population (which may cause convergence to a local, rather than global, optimum) and to fine-tune the solutions. Two forms of mutation are used here: uniform and non-uniform mutation. In both kinds of mutation, a randomly chosen chromosome (i.e., a random piece) of a randomly chosen candidate solution is replaced with a new, randomly generated value. In uniform mutation [12], the new value is allowed to be any legal value. This provides coarse adjustment of the solutions. In non-uniform mutation [12], the new value is taken from a smaller and smaller neighborhood of the original value. This provides fine tuning of the solutions. Let v_k be the k^{th} chromosome of the gene v. That is, v is one complete set of parameters, and k is the randomly chosen piece of the solution to be modified. Let l_k and u_k be lower and upper limits on v_k. For the t^{th} GA generation, non-uniform mutation will replace v_k with a new chromosome v_k', which is formed according to [12]

$$v_k^{'} = \begin{cases} v_k + \Delta(t, u_k - v_k), & d = 0 \\ v_k - \Delta(t, v_k - l_k), & d = 1 \end{cases} \quad \textbf{(31)}$$

where *d* is a random digit that specifies whether to increase or decrease the chromosome. The function *D(t,y)* returns a value in the interval [*0*, *y*] and is defined as [12]

$$\Delta(t, y) = y(1 - r^{(1-t/T)^b}) \quad \textbf{(32)}$$

where *T* is the total number of GA generations to be run, *b* is a parameter that specifies how fast the function *D(t,y)* should converge to 0, and *r* is a random number between 0 and 1. The probablilty that *D(t,y)* is close to 0 increases as *t* increases [12]. If *t* equals *T* (that is, if the GA is performing its last generation), the function *D(t,y)* equals 0. In other words, the function converges to 0 as the GA generations progress. The non-uniform mutation operator is useful because it allows a coarse search at first (when $t << T$), but gradually narrows the search as the algorithm runs. This allows fine local tuning of the solutions [12].

4.3 Customizing the Genetic Algorithm for OPF

In order to improve its convergence, the GA was customized for the OPF problem. Many of the strategies presented here were found by trial and error.

4.3.1 General GA parameters

The GA was run with a population size of 20 candidate solutions. The population was allowed to evolve for 10 generations. Elitism is used to guarantee that the best 5% of the population survives into the next generation. Some researchers evolve the population until the population becomes homogeneous (or nearly so). However, in this project, evolution progresses for a fixed number of generations.

4.3.2 Accounting for Static-VAR compensation

If the static-VAR compensation has changed, a load-flow solution is required to get an exact answer. However, performing load-flow solutions is time-consuming and

therefore undesirable. Thus, to save time, the effects of the static-VAR compensation are approximated through Equation (22), which uses the Jacobian to approximate the effects of a change in reactive power on the states. This approximation is not accurate enough to distinguish between two solutions of similar quality. Thus, the approximation is sufficient to determine which solutions are of poor quality and which are promising, but a fast-decoupled load flow must be used to determine the exact effect of the VAR compensation on the good-quality solutions. Specifically, the FDLF is used on any solution whose fitness is at least half as good as the best solution found by the algorithm and if its penalty is less than $10. This allows the algorithm to use the FDLF algorithm on the high-quality solutions that need it without wasting time on poor solutions.

4.3.3 Re-calibrating the linearization of the load-flow equations

Since the load-flow Jacobian is a linearized matrix, it is necessary to update the Jacobian if the GA's best solution has changed significantly. Recall that all members of the GA population (that is, all candidate solutions) are defined in terms of their difference with the best solution. Thus, whenever a new solution is found that improves the fitness by at least 1%, the load-flow Jacobian, rectangular submatrix, and nullspace are recalculated. The candidate solutions are then projected onto the new nullspace. This projection is accomplished in several steps. First, the best solution in the population is chosen as the reference solution. Its state vector is used to compute the Jacobian, and all other solutions are defined with respect to this reference. For every candidate solution, the old nullspace is used to convert the nullspace coefficients into a corresponding state vector. This state vector is substituted into the load flow equations to get the resulting real and reactive power at each generator. Because modeling errors resulting from the linearization inherent in computing the Jacobian, the real and reactive power at the load buses may not be exactly at their required values—particularly if the state vector varies greatly from the reference state vector used in computing the Jacobian. Even small changes in the states can lead to significant changes in power. To counteract this error, the load bus real and reactive powers are re-set to their required values. The new real and reactive powers are then input to a standard load-flow program to find the resulting, new state vector. The difference between the new state vector and the reference state vector is

then projected onto the nullspace, which gives the updated list of nullspace coefficients for the GA population.

4.3.4 Seeding the initial GA population

In theory, the GA should be able to converge from a completely random set of initial guesses (random initial population)—if the GA is allowed to evolve for enough generations [13]. However, convergence is hastened if any prior knowledge of the problem is incorporated into the algorithm [12,13]. One of the contributions of this work is to speed convergence by not wasting time solving load-flow equations. Because of the nullspace method employed in this work, the power at load buses is never altered (to the extent that the linearization is accurate). Therefore, the initial reference guess is required to have the correct power at the load buses. This can be accomplished either by solving for the reference state via a load-flow solution or by using a state vector that is known to satisfy the load bus power requirements.

In this work, the population is seeded with initial solutions given in the literature. The 30-bus system is seeded with the initial solution used by Alsac and Stott [10]. The 118-bus system is seeded with the state vector similar to the one given by Reid and Hasdorff [18]. Reid and Hasdorff do not say what their transformer settings are. Using the settings from the University of Washington [17], the vector in [18] does not meet all of the power requirements at the buses. That is, using this starting vector in the load flow equations gives real and reactive power at the buses that do not equal their specified values. To correct for this discrepancy, the load bus power values are reset to their specified values. These corrected power values are then used in a load flow solution, which then provides the corresponding voltage magnitude and angle at the buses.

4.4 Applying the Load-flow Equations

In order to apply the load-flow equations efficiently, a relationship is derived to account for changes in transformer tap settings without recomputing the relevant quantities from scratch. Moreover, some convergence issues are addressed.

4.4.1 Adjusting the equations for changes in transformer taps

In order to account for changes in transformer taps, the first step is to update the system Y_{bus} matrix, as described in Section 2.1.2.

Next, it is necessary to update the load-flow Jacobian. As with the Y_{bus} matrix, it is possible—but not desirable—to recompute the Jacobian from scratch each time a tap setting is changed. Instead, a contribution of this work is the derivation of the tap settings' effect on the Jacobian. As noted in Section 2.1.1, changing one transformer's tap setting alters a 4×4 submatrix of the Jacobian. Let this submatrix be partitioned into four 2×2 submatrices:

$$\Delta J_{4\times 4} = \begin{bmatrix} \Delta J_{11} & \Delta J_{12} \\ \Delta J_{21} & \Delta J_{22} \end{bmatrix} \quad \textbf{(33)}$$

To calculate DJ_{4x4}, we change one transformer tap setting at a time and subtract the old Jacobian from the new. The matrix DJ_{4x4} will be 0 at the positions of J not affected by the transformer. The only elements of J affected by a transformer are those elements that depend on the Y_{BUS} elements connected to the transformer. Thus, the change in J will depend on the changes in Y_{BUS}.

Recall from Section 2.1.2 that

$$\Delta Y_{12} = (t_0 - t)Y_L \quad \textbf{(34)}$$

Let DG and DB be defined respectively as the real and imaginary parts of DY_{12}. Similarly, let DG_{PP} and DB_{PP} be defined respectively as the real and imaginary parts of DY_{PP}. Define V_P and V_S respectively as the voltage magnitude at the primary and secondary of the transformer. Similarly, define d_P and d_S as the corresponding voltage angles. For convenience, define

$$G_S = \Delta G \sin(d_P - d_S) \quad \textbf{(35)}$$

$$G_C = \Delta G \cos(d_P - d_S) \quad \textbf{(36)}$$

$$B_S = \Delta B \sin(d_P - d_S) \quad \textbf{(37)}$$

$$B_C = \Delta B \cos(d_P - d_S) \quad \textbf{(38)}$$

where the subscripts attached to *G* and *B* (that is, *S* or *C*) refer to whether the variables are defined in terms of the *sine* or *cosine* of the difference in angle at the primary and secondary.

The submatrices in Equation (33) are found to be

$$\Delta J_{11} = V_P V_S \begin{bmatrix} -G_S + B_C & G_S - B_C \\ -G_S - B_C & G_S + B_C \end{bmatrix} \quad \textbf{(39)}$$

$$\Delta J_{12} = \begin{bmatrix} V_S(G_C + B_S) + 2V_P G_{PP} & V_P(G_C + B_S) \\ V_S(G_C - B_S) & V_P(G_C - B_S) \end{bmatrix} \quad \textbf{(40)}$$

$$\Delta J_{21} = V_P V_S \begin{bmatrix} G_C + B_S & -G_C - B_S \\ -G_C + B_S & G_C - B_S \end{bmatrix} \quad \textbf{(41)}$$

$$\Delta J_{22} = \begin{bmatrix} V_S(G_S - B_C) - 2V_P B_{PP} & V_P(G_S - B_C) \\ V_S(-G_S - B_C) & V_P(-G_S - B_C) \end{bmatrix} \quad \textbf{(42)}$$

4.4.2 Achieving convergence

As already stated, the FDLF has a wider region of convergence than the Newton-Raphson method [1]. These relative convergence characteristics were observed for the IEEE 118-bus system. For the 118-bus system, the Newton-Raphson method failed to converge. Instead, it gave unrealistic voltage values such as 10^5 p.u. However, the FDLF did converge for this system.

Sometimes, during the course of updating the Jacobian's reference state, a reasonably good solution was observed to undergo sudden reductions in its fitness. Upon further inspection, it was discovered that 2π radians was added to or subtracted from some

voltage angles in the course of the load-flow solution. Although this alteration would not affect the power flows, it would degrade the accuracy of the linearization, since the changes in angle are no longer small. Since the algorithm depends on linearization, this loss of accuarcy in the linearization caused the affected candidate solutions to become corrupted, hindering their fitness. Mapping the angles to the interval $[-\pi,\pi]$ radians alleviated the problem.

Chapter 5. Results

The GA-OPF algorithm is demonstrated on the three test cases described earlier: Steinberg and Smith's example, the IEEE 30-bus system, and the IEEE 118-bus system. Furthermore, since electric utilities are required to meet stricter and stricter environmental constraints, the cost of meeting emission constraints is examined with the 118-bus system. The algorithm is programmed in the Math Works' *Matlab* computation environment and run on a 300-MHz Pentium II computer.

5.1 Steinberg and Smith's Example

Steinberg and Smith's example demonstrates the inadequacy of setting incremental cost equal to each other when the incremental cost curves are not monotonic. To illustrate this, Steinberg and Smith provide a parametric graph [4] of generation cost for all combinations of output for each machine. For simplicity, only the optimal loading is given here, in Figure 7. The solution was found by a Genetic Algorithm and agrees with Steinberg and Smith's graph [4]. For the optimal solution, the machines are not necessarily operated at equal heat rates. The optimal heat rates of the machines are given in Figure 8.

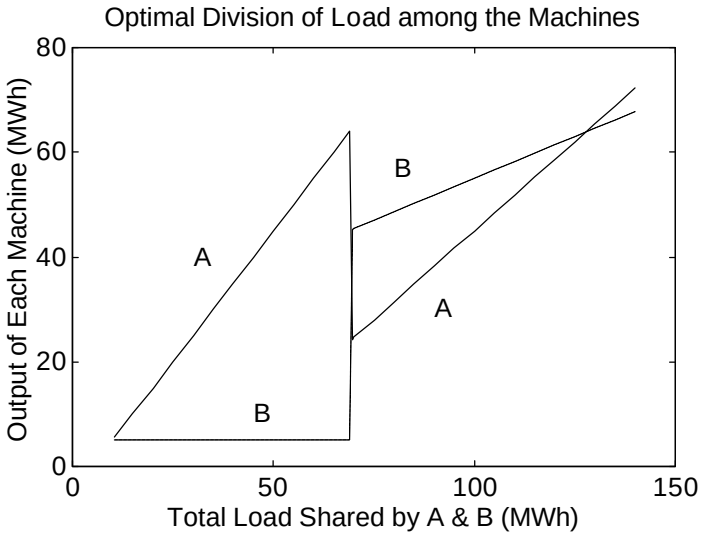

Figure 7. Optimal loading for Steinberg and Smith's example

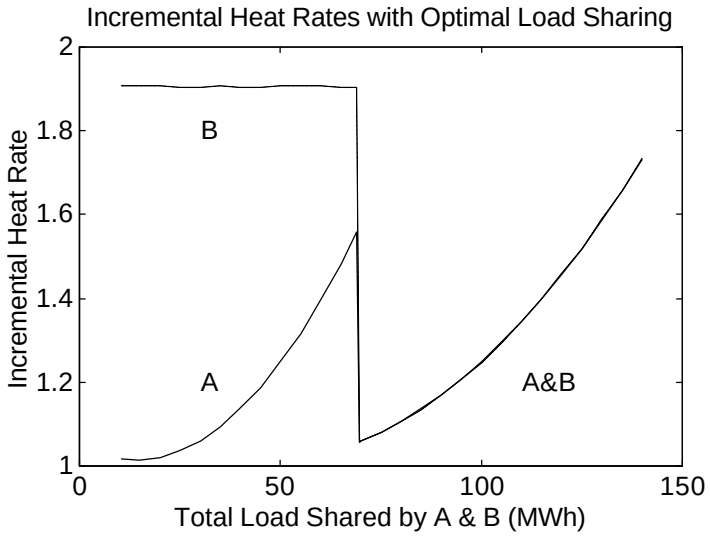

Figure 8. Optimal heat rates in Steinberg and Smith's example

At first, Machine A supplies all new load, while Machine B is held at its minimum value. Since Machine B's incremental heat rate is very large for small loads, it is cheaper for Machine A to pick up the new load—for a while. For loads larger than a threshold of about 69 MWh, Machine B supplies a substantial portion of the load. At the threshold, both machines' outputs are suddenly changed. For loads above the threshold, the machines are operated with equal incremental heat rates, as in traditional economic dispatch. For these larger loads, Machine A has become so expensive to operate that it makes economic sense to allow Machine B to supply some of the load. Of course, an electric power plant cannot change its output instantaneously, so a rate limit would have to be applied to the output in practice.

Above the threshold, both machines operate at a lower heat rate than did Machine A immediately below the threshold. One oddity of this example is that the system heat rate graph is discontinuous. For small loads, the system heat rate equals the heat rate of Machine A. Since Machine A is supplying all new increments of load, the system's cost for the new load equals Machine A's cost. For large loads, both machines are supplying new increments of load and have the same heat rate, which is also the heat rate for the system. The system's incremental heat rate is given in Figure 9.

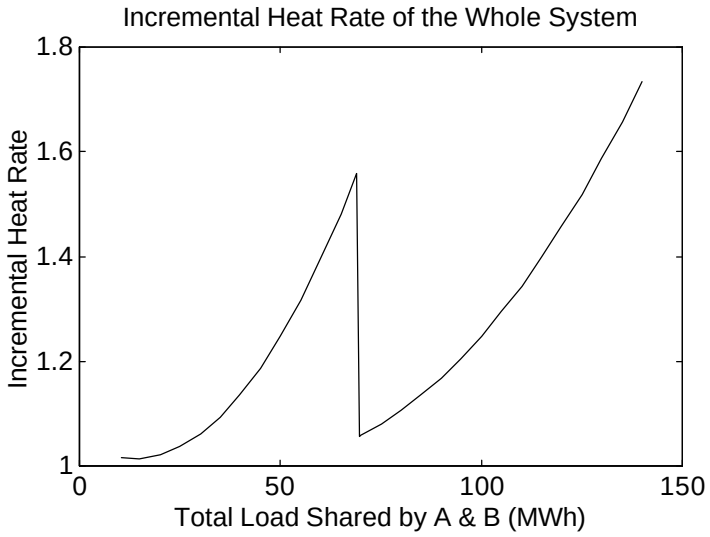

Figure 9. System heat rate, with optimal load sharing, in Steinberg and Smith's example

Thus, because of the discontinuity at 69 MWh, it is more expensive to supply the 65th MWh than the 75th MWh.

5.2 IEEE 30-bus system

Alsac and Stott's paper [10] provides a quantitative benchmark to demonstrate the accuracy of the GA-OPF algorithm. Since line flows and NO_X are unconstrained in [10], these quantities are unconstrained here also, to allow direct comparison of the results, as in Table 1:

Quantity	Alsac and Stott [10]	GA-OPF
Cost per hour	$802	$806
P(1)	1.76	1.70
P(2)	0.49	0.50
P(5)	0.22	0.20
P(8)	0.22	0.24
P(11)	0.12	0.11
P(13)	0.12	0.18

Table 1. Comparison of results for 30-bus system

Thus, the GA-OPF method was able to find a cost within 0.8% of that by Alsac and Stott. This demonstrates the algorithm's accuracy in finding an answer. The algorithm required approximately 15 minutes to converge.

5.3 IEEE 118-bus system

In order to demonstrate the algorithm on a more complicated system, the algorithm was run on the IEEE 118-bus system. First, the line limits and emissions were ignored. Then, both constraints were enforced. Because the 118-bus data had to be gathered from a variety of sources, it is not possible to compare these results directly to any other results. Reid and Hasdorff's voltage magnitudes and angles [18] are used to seed the initial GA population. Convergence requires approximately 2 hours.

5.3.1 Without Line Flow or Emission Constraints

The GA-OPF algorithm converges to a cost of $17,700/hr. For comparison with the constrained case, the line flows and emissions were calculated (but not constrained). The largest line flow is 347 MVA, and the total emissions are 34.8 lb/hour. As a very rough comparison, Reid and Hasdorff converge to a cost of $20,132/hr, using a different set of assumptions. Figure 10 and Figure 11 illustrate how much the GA-OPF algorithm altered the voltage magnitudes and angles, respectively. The light, broken curves represent the initial guess, and the dark, solid line is the GA-OPF's final answer.

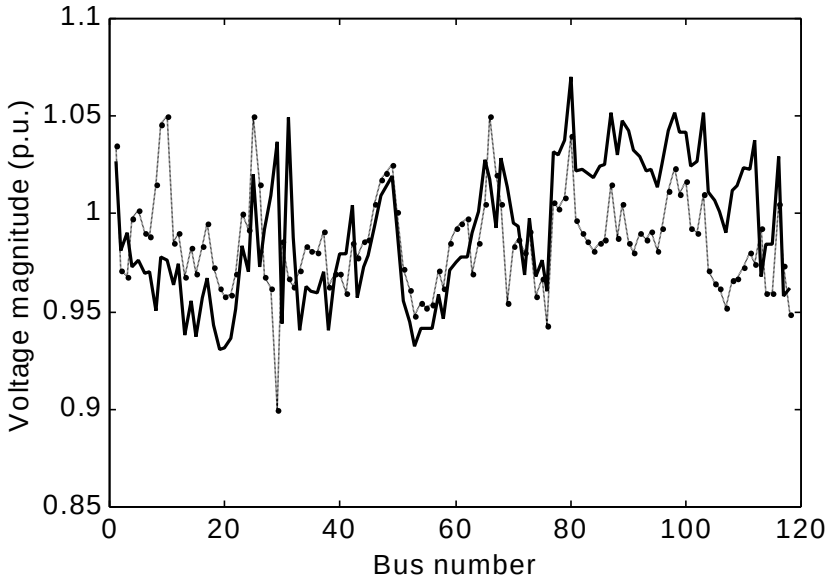

Figure 10. Voltage magnitude comparison, unconstrained line flows and emission

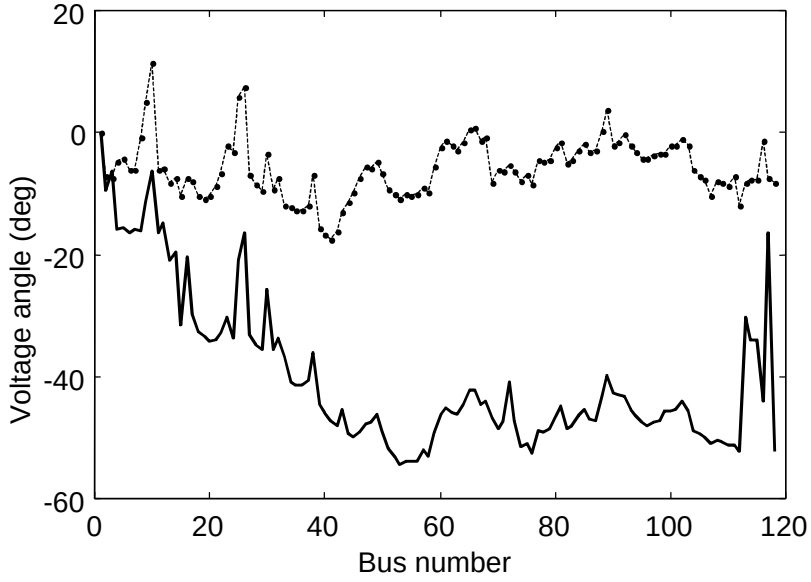

Figure 11. Voltage angle comparison, unconstrained line flows and emission

The voltage magnitudes did not change as much as the angles. This is understandable, since the VAR compensation and tap settings both alter the system's reactive power, which depends more strongly on voltage magnitude than on angle. In other words, these compensation devices can adjust themselves so that the reactive power changes without changing the voltage magnitude greatly. However, no such devices are assumed for real power. Thus, any alteration in real power causes the voltage angle to change.

5.3.2 With Line Flow and Emission Constraints

Here, the line flows are constrained to be less than 3 and NO_x is constrained to be less than 37.5 lb/hour. The GA-OPF algorithm found a solution with a cost of $18,900/hr, with NO_x of 37.1 lb/hour and a maximum line flow of 260 MVA.

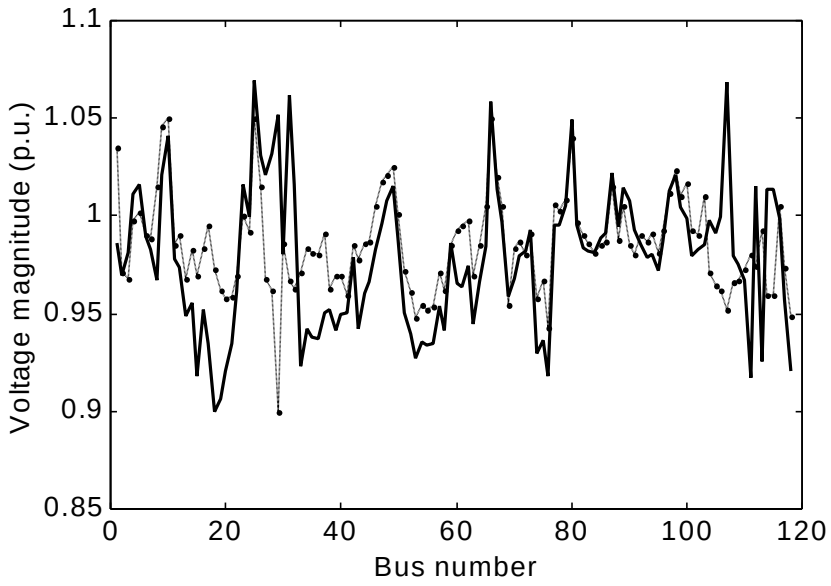

Figure 12. Voltage magnitude comparison, constrained line flows and emission

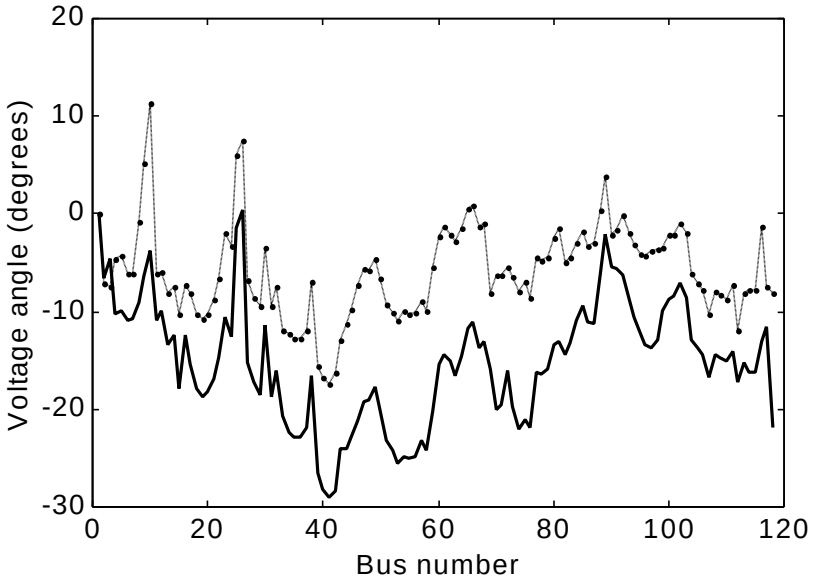

Figure 13. Voltage angle comparison, constrained line flows and emission

It is interesting to note that the angles are not altered as much in this case as in the unconstrained case.

Chapter 6. Conclusion

The GA has demonstrated its ability to solve the OPF problem. By avoiding the repeated solution of the load-flow equations, the unique chromosome encoding presented here improves execution time substantially. The mathematical derivation of the effects of the transformer taps on the Jacobian saves execution time by avoiding the recomputation of the entire matrix. By using linear algebra's nullspace theory to reduce the search space that must be examined, the algorithm spends less time evaluating illegal solutions. Without the nullspace theory, the algorithm would become overwhelmed with the sheer number of solutions that fail to meet the equality constraints for power at the load buses. Furthermore, by penalizing, rather than discarding, illegal solutions, the algorithm can glean useful information even from illegal solutions.

The GA-OPF method has shown its flexibility in that it allows incremental cost curves to have arbitrary shape. Whereas constraints hinder most traditional economic dispatch algorithms, the GA has demonstrated its ability to enforce constraints, even nonlinear constraints such as the presence of discrete control variables (such as tap-changing transformers and static-VAR compensators). Furthermore, the GA has demonstrated its ability to enforce environmental constraints.

Because of its flexibility both in enforcing a wide range of constraints and in its ability to optimize with an arbitrarily shaped cost-curve, the GA-OPF method is a promising method to solve the optimal power flow problem.

Chapter 7. Continued Research

The most logical continuation of this research focuses on improving the algorithm's execution time. Some possible ways to improve execution time are implementing the algorithm on a parallel computer, porting the algorithm to another computer language, using mathematical approximations for the equations, and using a nonlinear expression in place of the nullspace equation. A genetic algorithm is well suited to parallel computation, but the cost of a parallel computer is an obvious disadvantage to this option. Although Matlab is often a convenient language for development, other languages may execute the algorithm faster. A study could be conducted to investigate the relative performance of the algorithm when ported to various computer languages, such as *C*. Since a significant amount of computational effort is spent in solving the load-flow equations, approximating the load-flow equations by easier-to-solve expressions may save computation time. Finally, replacing the nullspace equation with a nonlinear expression may save time by reducing the required number of load-flow solutions. The existing OPF-GA algorithm must recalibrate itself by performing a load-flow solution on every member of the population whenever the fitness improves significantly, to ensure that the linearization is accurate. If the linearization step could be replaced with some convenient, nonlinear expression—one that is accurate for a larger domain of state variables—the recalibration step, which is computationally intensive, could be performed less often. Thus, there are several ways in which the GA-OPF method could be studied further.

Chapter 8. References

[1] A. S. Debs, *Modern Power Systems Control and Operation*, (Boston: Kluwer Academic Publishers, 1988)

[2] C. A. Gross, *Power System Analysis, second edition* (New York: John Wiley & Sons, 1986)

[3] Potomac Electric Power Company, "Power Plant Performance Monitoring and Improvement, Volume 2: Incremental Heat Rant Sensitivity Analysis," *EPRI CS/EL–4415*, Interim Report, February 1986

[4] M. J. Steinberg and T. H. Smith, *Economy Loading of Power Plants and Electric Systems*, (New York: John Wiley & Sons, 1943)

[5] K. Y. Lee, Y. M. Park, and J. L. Ortiz, "A United Approach to Optimal Real and Reactive Power Dispatch," *IEEE Trans. on Power Apparatus and Systems*, Vol. PAS-104, No. 5 (May 1985), pp. 1147–1153

[6] A. G. Bakirtzis, *Security Control Computations for Large Power Systems*, Ph.D. dissertation, Georgia Institute of Technology, November 1983

[7] K. P. Wong and Y. W. Wong, "Combined Genetic Algorithm / Simulated Annealing / Fuzzy Set Approach to Short-term Generation Scheduling with Take-or-pay Fuel Contract," *IEEE Trans. on Power Systems*, Vol. 11, No. 1 (February 1996), pp. 128–136

[8] A. Bakirtzis, V. Petridis, and S. Kazarlis, "Genetic Algorithm Solution to the Economic Dispatch Problem," *IEE Proceedings on Generation, Transmission, and Distribution*, Vol. 141, No. 4 (July 1994), pp. 377–382

[9] L. L. Lai and J. T. Ma, "Application of Evolutionary Programming to Reactive Power Planning—Comparison with Nonlinear Programming Approach," *IEEE Trans. on Power Systems*, Vol. 12, No. 1 (February 1997), pp. 198–206

[10] O. Alsac and B. Stott, "Optimal Load Flow with Steady-state Security," *IEEE Trans. on Power Apparatus and Systems*, Vol. PAS-93, No. 3 (May-June 1974), pp. 745–751

[11] B. Stott and O. Alsac, "Fast Decoupled Load Flow," *IEEE Trans. on Power Apparatus and Systems*, Vol. PAS-93, No. 3 (May-June 1974), pp. 859–864

[12] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs, third edition*, (New York: Springer-Verlag, 1996)

[13] L. Davis, editor, *Handbook of Genetic Algorithms* (New York: International Thompson Computer Press, 1996)

[14] J. J. Greffenstette, "Genetic Algorithms," *IEEE Expert*, Vol. 8, No. 5 (October 1993), pp. 5–8

[15] P. Wayer, "Genetic Algorithms," *Byte*, Vol. 16, No. 1 (January 1991), pp. 361–368

[16] G. Strang, *Linear Algebra and Its Applications, Third edition*, (San Diego: Harcourt Brace Jovanovich, 1988)

[17] University of Washington archive, web address www.ee.washington.edu (visited on September 15, 1999). The data are given in Appendix B.

[18] G. F. Reid and L. Hasdorff, "Economic Dispatch Using Quadratic Programming," *IEEE Trans. on Power Apparatus and Systems*, Vol. PAS-92, No. 6 (Nov.-Dec. 1973), pp. 2015–2023

[19] IEEE PES Power System Engineering Committee, "Current Issues in Operations Planning," *IEEE Trans. on Power Sys*, Vol. 7, No. 3 (Aug. 1992), pp. 1197–1210

Appendix A. IEEE 30-bus system data

A schematic for the IEEE 30-bus system is given in Figure A.1 [17].

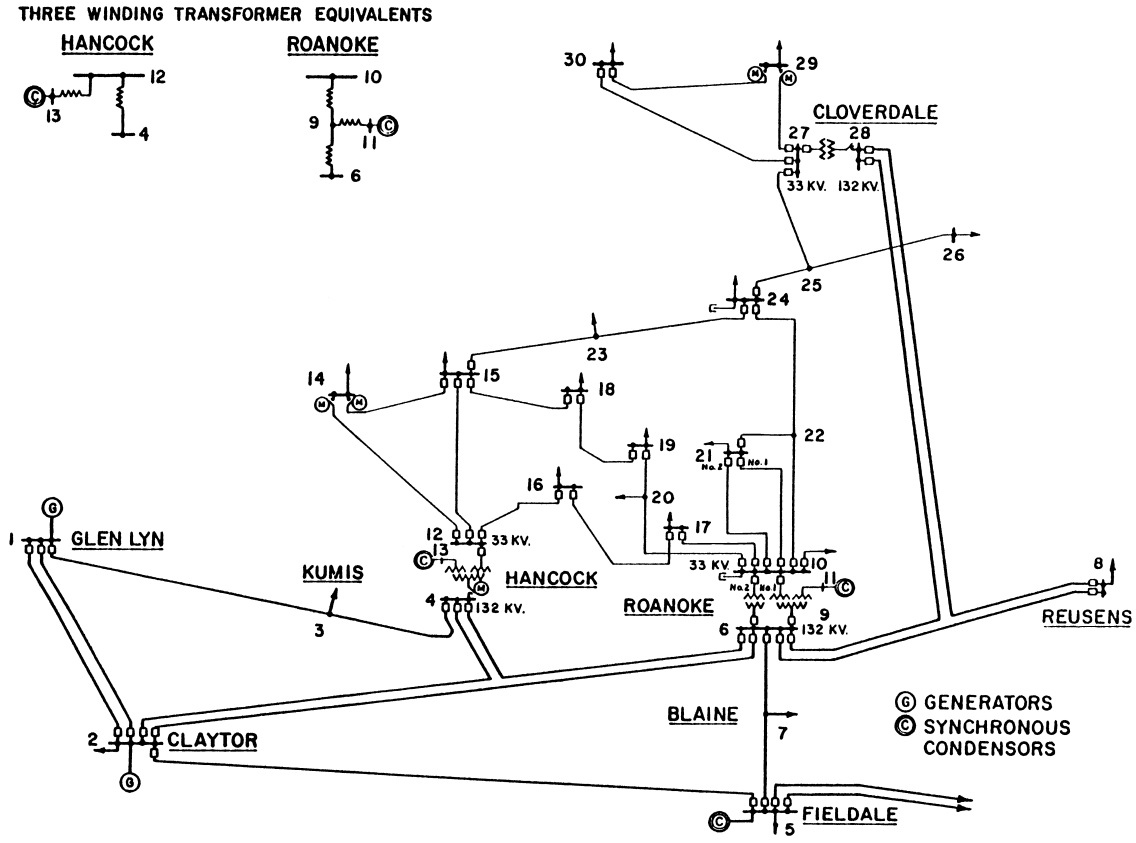

Figure A.1. IEEE 30-bus schematic

The data for the IEEE 30-bus system are taken from [10], which assumes that there are six generators. The six generators have quadratic cost curves, and the data are given in Table A.1. All power data are in per-unit, with a base of 100 MVA.

Bus	P_{min}	P_{max}	Q_{min}	Q_{max}	a	b	c
1	0.50	2.00	-0.2	2.5	0	200	37.5
2	0.20	0.80	-0.20	1.00	0	175	175.0
5	0.15	0.50	-0.15	0.80	0	100	625.0
8	0.10	0.35	-0.15	0.60	0	325	83.4
11	0.10	0.30	-0.10	0.50	0	300	250.0
13	0.12	0.40	-0.15	0.60	0	300	250.0

Table A.1. Generator data

The generator cost function is

$$C_T = \sum_{i=1}^{N} (a_i + b_i P_{Gi} + c_i P_{Gi}^2) \qquad \textbf{(A.1)}$$

where *N* equals 30, the number of buses, in this case.

The branch data and load data (in per-unit) are given in a data file, *30bus.dat*, which is listed below. In the file, any line beginning with a percent sign is a comment and is ignored. The file has a format in which a parameter name is followed by is value. Thus, if the string "R 0.01" appears in a line of the file, the resistance would be set to 0.01 p.u.

The lines beginning with the word *Bus* specify the bus number, the type of bus (Slack or PQ in this case, where PQ means a load bus), P=the real power load, and Q=the reactive power load. Since the load is specified for all of the buses, they are encoded in the data file as PQ, or load, buses. For some buses, a shunt susceptance (identified by *B*) is also described.

Any line that begins with the word *Line* specifies the parameters for a transmission line. The first two numbers are the numbers of the buses connected by the line. The remaining parameters are the R=series resistance, X=series reactance, Y=shunt admittance (assumed to be pure imaginary, that is a susceptance), and MVA=maximum power flow (in p.u.). The shunt admittance is divided equally between the ends of the

line; half of the specified number is assigned to each end. The letter *B* may also be used to represent the shunt susceptance.

Any line that begins with the word *Transformer* specifies a transmission line with a tap-changing transformer. The first two numbers are the primary and secondary buses, respectively. The remaining parameters are RL=series resistance, XL=series reactance, B=shunt susceptance, T=tap value, and MVA=maximum line flow. If *B* is not specified, it is assumed to be 0.

30bus.dat [10]

```
% IEEE 30-bus test case, given in per-unit (100 MVA base)
% The bus data give the P and Q LOAD at each bus, not the injected
% power. The optimal power flow program supplies the generation at the
% generation buses.

% Bus #1 has a load of 0 + j0 as well.
Bus  1 Slack  Angle 0
Bus  2  PQ  P 0.2170   Q 0.1270
Bus  3  PQ  P 0.0240   Q 0.0120
Bus  4  PQ  P 0.0760   Q 0.0160
Bus  5  PQ  P 0.9420   Q 0.1900
Bus  6  PQ  P 0        Q 0
Bus  7  PQ  P 0.2280   Q 0.1090
Bus  8  PQ  P 0.3000   Q 0.3000
Bus  9  PQ  P 0        Q 0
Bus 10  PQ  P 0.0580   Q 0.0200
Bus 11  PQ  P 0        Q 0
Bus 12  PQ  P 0.1120   Q 0.0750
Bus 13  PQ  P 0        Q 0
Bus 14  PQ  P 0.0620   Q 0.0160
Bus 15  PQ  P 0.0820   Q 0.0250
Bus 16  PQ  P 0.0350   Q 0.0180
Bus 17  PQ  P 0.0900   Q 0.0580
Bus 18  PQ  P 0.0320   Q 0.0090
Bus 19  PQ  P 0.0950   Q 0.0340
Bus 20  PQ  P 0.0220   Q 0.0070
Bus 21  PQ  P 0.1750   Q 0.1120
Bus 22  PQ  P 0        Q 0
Bus 23  PQ  P 0.0320   Q 0.0160
Bus 24  PQ  P 0.0870   Q 0.0670
Bus 25  PQ  P 0        Q 0
Bus 26  PQ  P 0.0350   Q 0.0230
Bus 27  PQ  P 0        Q 0
Bus 28  PQ  P 0        Q 0
Bus 29  PQ  P 0.0240   Q 0.0090
Bus 30  PQ  P 0.1060   Q 0.0190
```

```
Line  1  2 R .0192 X .0575 Y .0264  MVA 1.3
Line  1  3 R .0452 X .1852 Y .0204  MVA 1.3
Line  2  4 R .0570 X .1737 Y .0184  MVA .65
Line  3  4 R .0132 X .0379 Y .0042  MVA 1.3
Line  2  5 R .0472 X .1983 Y .0209  MVA 1.3
Line  2  6 R .0581 X .1763 Y .0187  MVA .65
Line  4  6 R .0119 X .0414 Y .0045  MVA .9
Line  5  7 R .0460 X .1160 Y .0102  MVA .7
Line  6  7 R .0267 X .0820 Y .0085  MVA 1.3
Line  6  8 R .0120 X .0420 Y .0045  MVA .32
Transformer 6  9 RL .0000 XL .2080 T 1.078  MVA .65
Transformer 6 10 RL .0000 XL .5560 T 1.069  MVA .32
Line  9 11 R .0000 X .2080  MVA .65
Line  9 10 R .0000 X .1100  MVA .65
Transformer 4 12 RL .0000 XL .2560 T 1.032  MVA .65
Line 12 13 R .0000 X .1400  MVA .65
Line 12 14 R .1231 X .2559  MVA .32
Line 12 15 R .0662 X .1304  MVA .32
Line 12 16 R .0945 X .1987  MVA .32
Line 14 15 R .2210 X .1997  MVA .16
Line 16 17 R .0824 X .1932  MVA .16
Line 15 18 R .1070 X .2185  MVA .16
Line 18 19 R .0639 X .1292  MVA .16
Line 19 20 R .0340 X .0680  MVA .32
Line 10 20 R .0936 X .2090  MVA .32
Line 10 17 R .0324 X .0845  MVA .32
Line 10 21 R .0348 X .0749  MVA .32
Line 10 22 R .0727 X .1499  MVA .32
Line 21 22 R .0116 X .0236  MVA .32
Line 15 23 R .1000 X .2020  MVA .16
Line 22 24 R .1150 X .1790  MVA .16
Line 23 24 R .1320 X .2700  MVA .16
Line 24 25 R .1885 X .3292  MVA .16
Line 25 26 R .2544 X .3800  MVA .16
Line 25 27 R .1093 X .2087  MVA .16
Transformer 28 27 RL .0000 XL .3960 T 1.068  MVA .65
Line 27 29 R .2198 X .4153  MVA .16
Line 27 30 R .3202 X .6027  MVA .16
Line 29 30 R .2399 X .4533  MVA .16
Line  8 28 R .0636 X .2000 Y .0214  MVA .32
Line  6 28 R .0169 X .0599 Y .0065  MVA .32
```

Appendix B. IEEE 118-bus system data

The IEEE 118-bus generator data are taken from [18] and given in Table B.1 (all power values are in per-unit):

Bus	P_{min}	P_{max}	Q_{min}	Q_{max}	a	b	c
1	1.0	7.0	-3.0	3.0	150	189	50
10	1.0	5.5	-1.47	2.0	115	200	55
12	0.1	3.5	-0.35	1.2	40	350	60
25	0.5	3.5	-0.47	1.4	122	315	55
26	1.0	4.5	-10.0	10.0	125	305	50
49	0.5	3.5	-0.85	2.1	120	275	70
59	0.5	3.0	-0.6	1.8	70	345	70
61	0.5	3.0	-1.0	3.0	70	345	70
65	0.5	5.0	-0.67	2.0	130	245	50
66	0.5	5.0	-0.67	2.0	130	245	50
80	0.5	5.5	-1.65	2.8	135	235	55
89	1.0	8.0	-2.1	3.0	200	160	45
100	0.5	3.5	-5.0	1.55	70	345	70
103	0	2.0	-0.6	0.6	45	328	60

Table B.1. Generator data, 118-bus system

The generator cost function is

$$C_T = \sum_{i=1}^{N} (a_i + b_i P_{Gi} + c_i P_{Gi}^2) \qquad \textbf{(B.1)}$$

where *N* equals 118, the number of buses, in this case.

The limits on static VAR compensation for the 118-bus system are taken from [18] and given in Table B.1 (all values are in per-unit):

Bus	Q_{cMin}	Q_{cMax}
4	-3.0	3.0
6	-0.6	0.6
15	-0.1	0.3
18	-0.6	0.6
19	-0.6	0.6
24	-3.0	3.0
27	-3.0	3.0
31	-3.0	3.0
32	-0.6	0.6
34	-0.6	0.6
36	-0.6	0.6
40	-3.0	3.0
42	-3.0	3.0
46	-1.0	1.0
54	-3.0	3.0
55	-0.6	0.6
56	-0.6	0.6
62	-0.2	0.2
69	-0.6	0.6
70	-0.6	0.6
72	-1.0	1.0
73	-1.0	1.0
74	-0.6	0.6
76	-0.6	0.6
77	-0.2	0.7
85	-0.6	0.6
87	-1.0	10.0
90	-3.0	3.0
91	-1.0	1.0
92	-0.6	0.6
99	-1.0	1.0
104	-0.6	0.6
105	-0.6	0.6
107	-2.0	2.0
110	-0.6	0.6
111	-1.0	10.0
112	-1.0	10.0
113	-1.0	2.0
116	-10.0	10.0

Table B.2. Static VAR limits, 118-bus system

A schematic for the 118-bus system is given in Figure B.1 [17]. Note that this schematic assumes a different placement of generators and VAR compensation than what is used here.

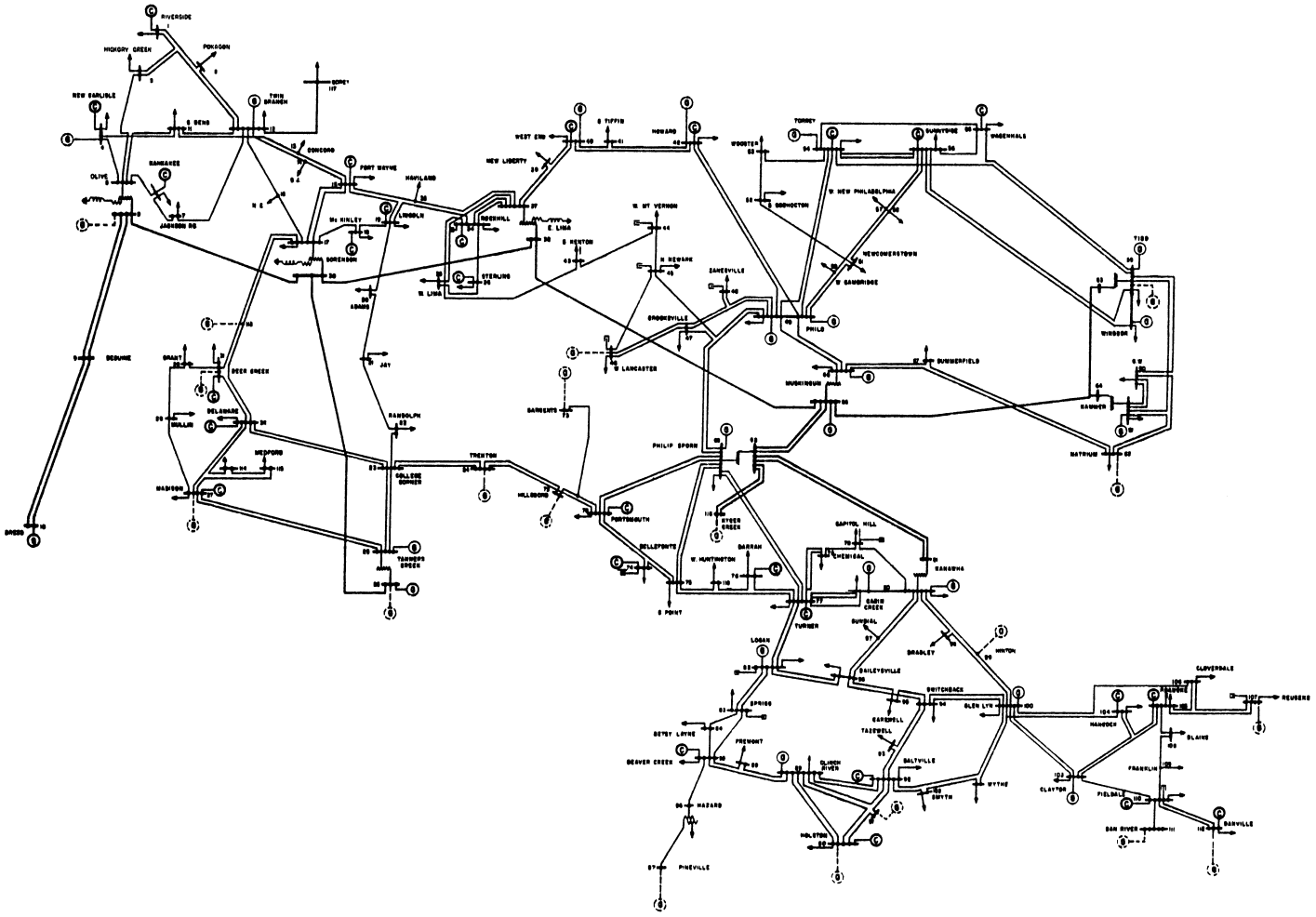

Figure B.1. IEEE 118-bus schematic

The line and bus data are given in *118bus.dat*, which is listed below. The format is identical to that of *30bus.dat*, which is described in Appendix A. The only differences are that the bus P and Q are given in MW or Mvars, rather than in per-unit.

118bus.dat [17]

```
% IEEE 118 Bus Test Case, from Univ. of Washington
% Per-unit base is 100 MVA.
% Impedance data are given in per-unit
% Bus P and Q data are given in MW or MVars.
% Bus 1 has a load of  P=51.0, Q=27.0

Bus 1   Slack  V 1.035 Angle 0
Bus 2   PQ  P  20.0  Q  9.0
Bus 3   PQ  P  39.0  Q  10.0
Bus 4   PQ  P  30.0  Q  12.0
Bus 5   PQ  P  0.0   Q  0.0    B  -0.40
Bus 6   PQ  P  52.0  Q  22.0
Bus 7   PQ  P  19.0  Q  2.0
Bus 8   PQ  P  0.0   Q  0.0
Bus 9   PQ  P  0.0   Q  0.0
Bus 10  PQ  P  0.0   Q  0.0
Bus 11  PQ  P  70.0  Q  23.0
Bus 12  PQ  P  47.0  Q  10.0
Bus 13  PQ  P  34.0  Q  16.0
Bus 14  PQ  P  14.0  Q  1.0
Bus 15  PQ  P  90.0  Q  30.0
Bus 16  PQ  P  25.0  Q  10.0
Bus 17  PQ  P  11.0  Q  3.0
Bus 18  PQ  P  60.0  Q  34.0
Bus 19  PQ  P  45.0  Q  25.0
Bus 20  PQ  P  18.0  Q  3.0
Bus 21  PQ  P  14.0  Q  8.0
Bus 22  PQ  P  10.0  Q  5.0
Bus 23  PQ  P  7.0   Q  3.0
Bus 24  PQ  P  0.0   Q  0.0
Bus 25  PQ  P  0.0   Q  0.0
Bus 26  PQ  P  0.0   Q  0.0
Bus 27  PQ  P  62.0  Q  13.0
Bus 28  PQ  P  17.0  Q  7.0
Bus 29  PQ  P  24.0  Q  4.0
Bus 30  PQ  P  0.0   Q  0.0
Bus 31  PQ  P  43.0  Q  27.0
Bus 32  PQ  P  59.0  Q  23.0
Bus 33  PQ  P  23.0  Q  9.0
Bus 34  PQ  P  59.0  Q  26.0  B  0.14
Bus 35  PQ  P  33.0  Q  9.0
Bus 36  PQ  P  31.0  Q  17.0
Bus 37  PQ  P  0.0   Q  0.0   B -0.25
Bus 38  PQ  P  0.0   Q  0.0
Bus 39  PQ  P  27.0  Q  11.0
Bus 40  PQ  P  20.0  Q  23.0
Bus 41  PQ  P  37.0  Q  10.0
Bus 42  PQ  P  37.0  Q  23.0
```

```
Bus 43  PQ  P  18.0  Q  7.0
Bus 44  PQ  P  16.0  Q  8.0   B  0.10
Bus 45  PQ  P  53.0  Q  22.0  B  0.10
Bus 46  PQ  P  28.0  Q  10.0  B  0.10
Bus 47  PQ  P  34.0  Q  0.0
Bus 48  PQ  P  20.0  Q  11.0  B  0.15
Bus 49  PQ  P  87.0  Q  30.0
Bus 50  PQ  P  17.0  Q  4.0
Bus 51  PQ  P  17.0  Q  8.0
Bus 52  PQ  P  18.0  Q  5.0
Bus 53  PQ  P  23.0  Q  11.0
Bus 54  PQ  P  113.0 Q  32.0
Bus 55  PQ  P  63.0  Q  22.0
Bus 56  PQ  P  84.0  Q  18.0
Bus 57  PQ  P  12.0  Q  3.0
Bus 58  PQ  P  12.0  Q  3.0
Bus 59  PQ  P  277.0 Q  113.0
Bus 60  PQ  P  78.0  Q  3.0
Bus 61  PQ  P  0.0   Q  0.0
Bus 62  PQ  P  77.0  Q  14.0
Bus 63  PQ  P  0.0   Q  0.0
Bus 64  PQ  P  0.0   Q  0.0
Bus 65  PQ  P  0.0   Q  0.0
Bus 66  PQ  P  39.0  Q  18.0
Bus 67  PQ  P  28.0  Q  7.0
Bus 68  PQ  P  0.0   Q  0.0
Bus 69  PQ  P  0.0   Q  0.0
Bus 70  PQ  P  66.0  Q  20.0
Bus 71  PQ  P  0.0   Q  0.0
Bus 72  PQ  P  0.0   Q  0.0
Bus 73  PQ  P  0.0   Q  0.0
Bus 74  PQ  P  68.0  Q  27.0   B  0.12
Bus 75  PQ  P  47.0  Q  11.0
Bus 76  PQ  P  68.0  Q  36.0
Bus 77  PQ  P  61.0  Q  28.0
Bus 78  PQ  P  71.0  Q  26.0
Bus 79  PQ  P  39.0  Q  32.0  B   0.20
Bus 80  PQ  P  130.0 Q  26.0
Bus 81  PQ  P  0.0   Q  0.0
Bus 82  PQ  P  54.0  Q  27.0  B   0.20
Bus 83  PQ  P  20.0  Q  10.0  B   0.10
Bus 84  PQ  P  11.0  Q  7.0
Bus 85  PQ  P  24.0  Q  15.0
Bus 86  PQ  P  21.0  Q  10.0
Bus 87  PQ  P  0.0   Q  0.0
Bus 88  PQ  P  48.0  Q  10.0
Bus 89  PQ  P  0.0   Q  0.0
Bus 90  PQ  P  78.0  Q  42.0
Bus 91  PQ  P  0.0   Q  0.0
Bus 92  PQ  P  65.0  Q  10.0
Bus 93  PQ  P  12.0  Q  7.0
Bus 94  PQ  P  30.0  Q  16.0
Bus 95  PQ  P  42.0  Q  31.0
Bus 96  PQ  P  38.0  Q  15.0
Bus 97  PQ  P  15.0  Q  9.0
Bus 98  PQ  P  34.0  Q  8.0
Bus 99  PQ  P  0.0   Q  0.0
```

```
Bus 100 PQ  P  37.0  Q  18.0
Bus 101 PQ  P  22.0  Q  15.0
Bus 102 PQ  P  5.0   Q  3.0
Bus 103 PQ  P  23.0  Q  16.0
Bus 104 PQ  P  38.0  Q  25.0
Bus 105 PQ  P  31.0  Q  26.0  B  0.20
Bus 106 PQ  P  43.0  Q  16.0
Bus 107 PQ  P  28.0  Q  12.0  B  0.06
Bus 108 PQ  P  2.0   Q  1.0
Bus 109 PQ  P  8.0   Q  3.0
Bus 110 PQ  P  39.0  Q  30.0  B  0.06
Bus 111 PQ  P  0.0   Q  0.0
Bus 112 PQ  P  25.0  Q  13.0
Bus 113 PQ  P  0.0   Q  0.0
Bus 114 PQ  P  8.0   Q  3.0
Bus 115 PQ  P  22.0  Q  7.0
Bus 116 PQ  P  0.0   Q  0.0
Bus 117 PQ  P  20.0  Q  8.0
Bus 118 PQ  P  33.0  Q  15.0

Line 1 2 R   0.03030  X   0.09990  B  0.02540  MVA 1.00
Line 1 3 R   0.01290  X   0.04240  B  0.01082  MVA 1.00
Line 4 5 R   0.00176  X   0.00798  B  0.00210  MVA 1.00
Line 3 5 R   0.02410  X   0.10800  B  0.02840  MVA 1.00
Line 5 6 R   0.01190  X   0.05400  B  0.01426  MVA 1.00
Line 6 7 R   0.00459  X   0.02080  B  0.00550  MVA 1.00
Line 8 9 R   0.00244  X   0.03050  B  1.16200  MVA 1.00
Transformer  8  5   R 0.00000  X   0.02670 B   0.0  T  0.985  MVA 1.00
Line 9 10    R  0.00258   X  0.03220  B  1.23000  MVA 1.00
Line 4 11    R  0.02090   X  0.06880  B  0.01748  MVA 1.00
Line 5 11    R  0.02030   X  0.06820  B  0.01738  MVA 1.00
Line 11  12  R  0.00595   X  0.01960  B  0.00502  MVA 1.00
Line 2 12    R  0.01870   X  0.06160  B  0.01572  MVA 1.00
Line 3 12    R  0.04840   X  0.16000  B  0.04060  MVA 1.00
Line 7 12    R  0.00862   X  0.03400  B  0.00874  MVA 1.00
Line 11  13  R  0.02225   X  0.07310  B  0.01876  MVA 1.00
Line 12  14  R  0.02150   X  0.07070  B  0.01816  MVA 1.00
Line 13  15  R  0.07440   X  0.24440  B  0.06268  MVA 1.00
Line 14  15  R  0.05950   X  0.19500  B  0.05020  MVA 1.00
Line 12  16  R  0.02120   X  0.08340  B  0.02140  MVA 1.00
Line 15  17  R  0.01320   X  0.04370  B  0.04440  MVA 1.00
Line 16  17  R  0.04540   X  0.18010  B  0.04660  MVA 1.00
Line 17  18  R  0.01230   X  0.05050  B  0.01298  MVA 1.00
Line 18  19  R  0.01119   X  0.04930  B  0.01142  MVA 1.00
Line 19  20  R  0.02520   X  0.11700  B  0.02980  MVA 1.00
Line 15  19  R  0.01200   X  0.03940  B  0.01010  MVA 1.00
Line 20  21  R  0.01830   X  0.08490  B  0.02160  MVA 1.00
Line 21  22  R  0.02090   X  0.09700  B  0.02460  MVA 1.00
Line 22  23  R  0.03420   X  0.15900  B  0.04040  MVA 1.00
Line 23  24  R  0.01350   X  0.04920  B  0.04980  MVA 1.00
Line 23  25  R  0.01560   X  0.08000  B  0.08640  MVA 1.00
Transformer  26 25  R 0.00000  X   0.03820 B   0.0  T  0.960  MVA 1.00
Line 25  27  R  0.03180   X  0.16300  B  0.17640  MVA 1.00
Line 27  28  R  0.01913   X  0.08550  B  0.02160  MVA 1.00
Line 28  29  R  0.02370   X  0.09430  B  0.02380  MVA 1.00
Transformer  30 17  R 0.00000  X   0.03880 B   0.0  T  0.960  MVA 1.00
Line 8 30    R  0.00431   X  0.05040  B  0.51400  MVA 1.00
```

```
Line 26  30  R  0.00799   X  0.08600  B  0.90800  MVA 1.00
Line 17  31  R  0.04740   X  0.15630  B  0.03990  MVA 1.00
Line 29  31  R  0.01080   X  0.03310  B  0.00830  MVA 1.00
Line 23  32  R  0.03170   X  0.11530  B  0.11730  MVA 1.00
Line 31  32  R  0.02980   X  0.09850  B  0.02510  MVA 1.00
Line 27  32  R  0.02290   X  0.07550  B  0.01926  MVA 1.00
Line 15  33  R  0.03800   X  0.12440  B  0.03194  MVA 1.00
Line 19  34  R  0.07520   X  0.24700  B  0.06320  MVA 1.00
Line 35  36  R  0.00224   X  0.01020  B  0.00268  MVA 1.00
Line 35  37  R  0.01100   X  0.04970  B  0.01318  MVA 1.00
Line 33  37  R  0.04150   X  0.14200  B  0.03660  MVA 1.00
Line 34  36  R  0.00871   X  0.02680  B  0.00568  MVA 1.00
Line 34  37  R  0.00256   X  0.00940  B  0.00984  MVA 1.00
Transformer  38 37  R 0.00000  X   0.03750 B   0.0  T  0.935  MVA 1.00
Line 37  39  R  0.03210   X  0.10600  B  0.02700  MVA 1.00
Line 37  40  R  0.05930   X  0.16800  B  0.04200  MVA 1.00
Line 30  38  R  0.00464   X  0.05400  B  0.42200  MVA 1.00
Line 39  40  R  0.01840   X  0.06050  B  0.01552  MVA 1.00
Line 40  41  R  0.01450   X  0.04870  B  0.01222  MVA 1.00
Line 40  42  R  0.05550   X  0.18300  B  0.04660  MVA 1.00
Line 41  42  R  0.04100   X  0.13500  B  0.03440  MVA 1.00
Line 43  44  R  0.06080   X  0.24540  B  0.06068  MVA 1.00
Line 34  43  R  0.04130   X  0.16810  B  0.04226  MVA 1.00
Line 44  45  R  0.02240   X  0.09010  B  0.02240  MVA 1.00
Line 45  46  R  0.04000   X  0.13560  B  0.03320  MVA 1.00
Line 46  47  R  0.03800   X  0.12700  B  0.03160  MVA 1.00
Line 46  48  R  0.06010   X  0.18900  B  0.04720  MVA 1.00
Line 47  49  R  0.01910   X  0.06250  B  0.01604  MVA 1.00
Line 42  49  R  0.07150   X  0.32300  B  0.08600  MVA 1.00
Line 42  49  R  0.07150   X  0.32300  B  0.08600  MVA 1.00
Line 45  49  R  0.06840   X  0.18600  B  0.04440  MVA 1.00
Line 48  49  R  0.01790   X  0.05050  B  0.01258  MVA 1.00
Line 49  50  R  0.02670   X  0.07520  B  0.01874  MVA 1.00
Line 49  51  R  0.04860   X  0.13700  B  0.03420  MVA 1.00
Line 51  52  R  0.02030   X  0.05880  B  0.01396  MVA 1.00
Line 52  53  R  0.04050   X  0.16350  B  0.04058  MVA 1.00
Line 53  54  R  0.02630   X  0.12200  B  0.03100  MVA 1.00
Line 49  54  R  0.07300   X  0.28900  B  0.07380  MVA 1.00
Line 49  54  R  0.08690   X  0.29100  B  0.07300  MVA 1.00
Line 54  55  R  0.01690   X  0.07070  B  0.02020  MVA 1.00
Line 54  56  R  0.00275   X  0.00955  B  0.00732  MVA 1.00
Line 55  56  R  0.00488   X  0.01510  B  0.00374  MVA 1.00
Line 56  57  R  0.03430   X  0.09660  B  0.02420  MVA 1.00
Line 50  57  R  0.04740   X  0.13400  B  0.03320  MVA 1.00
Line 56  58  R  0.03430   X  0.09660  B  0.02420  MVA 1.00
Line 51  58  R  0.02550   X  0.07190  B  0.01788  MVA 1.00
Line 54  59  R  0.05030   X  0.22930  B  0.05980  MVA 1.00
Line 56  59  R  0.08250   X  0.25100  B  0.05690  MVA 1.00
Line 56  59  R  0.08030   X  0.23900  B  0.05360  MVA 1.00
Line 55  59  R  0.04739   X  0.21580  B  0.05646  MVA 1.00
Line 59  60  R  0.03170   X  0.14500  B  0.03760  MVA 1.00
Line 59  61  R  0.03280   X  0.15000  B  0.03880  MVA 1.00
Line 60  61  R  0.00264   X  0.01350  B  0.01456  MVA 1.00
Line 60  62  R  0.01230   X  0.05610  B  0.01468  MVA 1.00
Line 61  62  R  0.00824   X  0.03760  B  0.00980  MVA 1.00
Transformer  63 59  R 0.00000  X   0.03860 B   0.0  T  0.960  MVA 1.00
Line 63  64  R  0.00172   X  0.02000  B  0.21600  MVA 1.00
```

```
Transformer  64 61  R 0.00000  X   0.02680 B   0.0  T  0.985  MVA 1.00
Line 38  65  R  0.00901   X  0.09860  B  1.04600  MVA 1.00
Line 64  65  R  0.00269   X  0.03020  B  0.38000  MVA 1.00
Line 49  66  R  0.01800   X  0.09190  B  0.02480  MVA 1.00
Line 49  66  R  0.01800   X  0.09190  B  0.02480  MVA 1.00
Line 62  66  R  0.04820   X  0.21800  B  0.05780  MVA 1.00
Line 62  67  R  0.02580   X  0.11700  B  0.03100  MVA 1.00
Transformer  65 66  R 0.00000  X   0.03700 B   0.0  T  0.935  MVA 1.00
Line 66  67  R  0.02240   X  0.10150  B  0.02682  MVA 1.00
Line 65  68  R  0.00138   X  0.01600  B  0.63800  MVA 1.00
Line 47  69  R  0.08440   X  0.27780  B  0.07092  MVA 1.00
Line 49  69  R  0.09850   X  0.32400  B  0.08280  MVA 1.00
Transformer  68 69  R 0.00000  X   0.03700 B   0.0  T  0.935  MVA 1.00
Line 69  70  R  0.03000   X  0.12700  B  0.12200  MVA 1.00
Line 24  70  R  0.00221   X  0.41150  B  0.10198  MVA 1.00
Line 70  71  R  0.00882   X  0.03550  B  0.00878  MVA 1.00
Line 24  72  R  0.04880   X  0.19600  B  0.04880  MVA 1.00
Line 71  72  R  0.04460   X  0.18000  B  0.04444  MVA 1.00
Line 71  73  R  0.00866   X  0.04540  B  0.01178  MVA 1.00
Line 70  74  R  0.04010   X  0.13230  B  0.03368  MVA 1.00
Line 70  75  R  0.04280   X  0.14100  B  0.03600  MVA 1.00
Line 69  75  R  0.04050   X  0.12200  B  0.12400  MVA 1.00
Line 74  75  R  0.01230   X  0.04060  B  0.01034  MVA 1.00
Line 76  77  R  0.04440   X  0.14800  B  0.03680  MVA 1.00
Line 69  77  R  0.03090   X  0.10100  B  0.10380  MVA 1.00
Line 75  77  R  0.06010   X  0.19990  B  0.04978  MVA 1.00
Line 77  78  R  0.00376   X  0.01240  B  0.01264  MVA 1.00
Line 78  79  R  0.00546   X  0.02440  B  0.00648  MVA 1.00
Line 77  80  R  0.01700   X  0.04850  B  0.04720  MVA 1.00
Line 77  80  R  0.02940   X  0.10500  B  0.02280  MVA 1.00
Line 79  80  R  0.01560   X  0.07040  B  0.01870  MVA 1.00
Line 68  81  R  0.00175   X  0.02020  B  0.80800  MVA 1.00
Transformer  81 80  R 0.00000  X   0.03700 B   0.0  T  0.935  MVA 1.00
Line 77  82  R  0.02980   X  0.08530  B  0.08174  MVA 1.00
Line 82  83  R  0.01120   X  0.03665  B  0.03796  MVA 1.00
Line 83  84  R  0.06250   X  0.13200  B  0.02580  MVA 1.00
Line 83  85  R  0.04300   X  0.14800  B  0.03480  MVA 1.00
Line 84  85  R  0.03020   X  0.06410  B  0.01234  MVA 1.00
Line 85  86  R  0.03500   X  0.12300  B  0.02760  MVA 1.00
Line 86  87  R  0.02828   X  0.20740  B  0.04450  MVA 1.00
Line 85  88  R  0.02000   X  0.10200  B  0.02760  MVA 1.00
Line 85  89  R  0.02390   X  0.17300  B  0.04700  MVA 1.00
Line 88  89  R  0.01390   X  0.07120  B  0.01934  MVA 1.00
Line 89  90  R  0.05180   X  0.18800  B  0.05280  MVA 1.00
Line 89  90  R  0.02380   X  0.09970  B  0.10600  MVA 1.00
Line 90  91  R  0.02540   X  0.08360  B  0.02140  MVA 1.00
Line 89  92  R  0.00990   X  0.05050  B  0.05480  MVA 1.00
Line 89  92  R  0.03930   X  0.15810  B  0.04140  MVA 1.00
Line 91  92  R  0.03870   X  0.12720  B  0.03268  MVA 1.00
Line 92  93  R  0.02580   X  0.08480  B  0.02180  MVA 1.00
Line 92  94  R  0.04810   X  0.15800  B  0.04060  MVA 1.00
Line 93  94  R  0.02230   X  0.07320  B  0.01876  MVA 1.00
Line 94  95  R  0.01320   X  0.04340  B  0.01110  MVA 1.00
Line 80  96  R  0.03560   X  0.18200  B  0.04940  MVA 1.00
Line 82  96  R  0.01620   X  0.05300  B  0.05440  MVA 1.00
Line 94  96  R  0.02690   X  0.08690  B  0.02300  MVA 1.00
Line 80  97  R  0.01830   X  0.09340  B  0.02540  MVA 1.00
```

```
Line 80  98  R  0.02380   X  0.10800  B  0.02860  MVA 1.00
Line 80  99  R  0.04540   X  0.20600  B  0.05460  MVA 1.00
Line 92  100 R  0.06480   X  0.29500  B  0.04720  MVA 1.00
Line 94  100 R  0.01780   X  0.05800  B  0.06040  MVA 1.00
Line 95  96  R  0.01710   X  0.05470  B  0.01474  MVA 1.00
Line 96  97  R  0.01730   X  0.08850  B  0.02400  MVA 1.00
Line 98  100 R  0.03970   X  0.17900  B  0.04760  MVA 1.00
Line 99  100 R  0.01800   X  0.08130  B  0.02160  MVA 1.00
Line 100 101 R  0.02770   X  0.12620  B  0.03280  MVA 1.00
Line 92  102 R  0.01230   X  0.05590  B  0.01464  MVA 1.00
Line 101 102 R  0.02460   X  0.11200  B  0.02940  MVA 1.00
Line 100 103 R  0.01600   X  0.05250  B  0.05360  MVA 1.00
Line 100 104 R  0.04510   X  0.20400  B  0.05410  MVA 1.00
Line 103 104 R  0.04660   X  0.15840  B  0.04070  MVA 1.00
Line 103 105 R  0.05350   X  0.16250  B  0.04080  MVA 1.00
Line 100 106 R  0.06050   X  0.22900  B  0.06200  MVA 1.00
Line 104 105 R  0.00994   X  0.03780  B  0.00986  MVA 1.00
Line 105 106 R  0.01400   X  0.05470  B  0.01434  MVA 1.00
Line 105 107 R  0.05300   X  0.18300  B  0.04720  MVA 1.00
Line 105 108 R  0.02610   X  0.07030  B  0.01844  MVA 1.00
Line 106 107 R  0.05300   X  0.18300  B  0.04720  MVA 1.00
Line 108 109 R  0.01050   X  0.02880  B  0.00760  MVA 1.00
Line 103 110 R  0.03906   X  0.18130  B  0.04610  MVA 1.00
Line 109 110 R  0.02780   X  0.07620  B  0.02020  MVA 1.00
Line 110 111 R  0.02200   X  0.07550  B  0.02000  MVA 1.00
Line 110 112 R  0.02470   X  0.06400  B  0.06200  MVA 1.00
Line 17  113 R  0.00913   X  0.03010  B  0.00768  MVA 1.00
Line 32  113 R  0.06150   X  0.20300  B  0.05180  MVA 1.00
Line 32  114 R  0.01350   X  0.06120  B  0.01628  MVA 1.00
Line 27  115 R  0.01640   X  0.07410  B  0.01972  MVA 1.00
Line 114 115 R  0.00230   X  0.01040  B  0.00276  MVA 1.00
Line 68  116 R  0.00034   X  0.00405  B  0.16400  MVA 1.00
Line 12  117 R  0.03290   X  0.14000  B  0.03580  MVA 1.00
Line 75  118 R  0.01450   X  0.04810  B  0.01198  MVA 1.00
Line 76  118 R  0.01640   X  0.05440  B  0.01356  MVA 1.00
```

Appendix C. Program Listings

This appendix lists the program code used in implementing the GA-OPF algorithm. The code is written for the Math Works' *Matlab* computation environment, version 5.0. The files *Init30.m* and *Init118.m* initialize the data for the 30-bus and 118-bus systems, respectively. These files define the Y-bus matrix, the cost functions, etc. The file *PwrData.m* is used to read the power data from the data files (*30bus.dat* or *118bus.dat*, which are given in Appendices A and B). *Start118.m* is a script file that defines the starting point for the 118-bus system.

The file *OPF_NXGA.m* performs the actual genetic algorithm. It contains two nested loops, an outer loop that represents the GA generations and an inner loop that processes each candidate solution in the population. At the end of each generation, the algorithm performs the genetic operators to randomly selected members of the population.

The remaining files support the work of *OPF_NXGA*. *Fitness.m* computes the fitness and penalty of a given candidate solution. The file *J_NewTaps* computes the new Jacobian when the tap settings have changes. The load-flow equations are implemented by *LF_Eqs.m*, and the load-flow Jacobian is computed by *LF_Jacob.m*.

The traditional Newton-Raphson Optimal Power Flow algorithm is implemented by *OPF.m*, which calls *OPF_Jacb.m* to compute its Jacobian. The Fast-Decoupled load-flow is implemented by *FDLF.m*.

C.1 Init30.m (Initialize 30-bus system)

```
% Define the fundamental constraints and other constants as global variables
global N Nc NumGenU CostCoeff NOxCoeff GenInd QcInd AMin AMax
global VmMin VmMax PMin PMax QMin QMax QcMin QcMax NOxMax Ptol LineMVA

[Ybus,NodeList,BusTypes,Pd,Qd,Vg,SlackAng,LineMVA,Xform]=PwrData('30bus.dat');
LineMVA = LineMVA + diag(inf*ones(length(NodeList),1));  % Don't limit shunt power at buses

VmLim=ones(30,1)*[.95 1.05];
VmLim([2 5 8 11 13],2)=1.1;
MVA = 100; % MVA base for p.u.
N=30;
SlackInd=find(BusTypes==1);
```

```
% Define the generation limits (converted to p.u.)
GenInd = [1 2 5 8 11 13];
PLim=zeros(30,2); QLim=zeros(30,2);
PLim(GenInd,:) = [50 200; 20 80; 15 50; 10 35; 10 30; 12 40]/MVA;
QLim(GenInd,:) = [-30 200; -20 100; -15 80; -15 60; -10 50; -15 60]/MVA;

QcInd = [10 24];      % Indices used by Alsac
QcLim = [0 .5; 0 .5];

TInd = Xform(:,1:2);
TLim = ones(length(TInd),1)*[.9 1.1];
OTaps = Xform(:,3);
% OTaps = Old Taps.  These are the nominal taps values specified in the
%    data file.

% Do an ordinary load flow to set the initial guess
PgList=zeros(N,1);QgList=zeros(N,1);QcList=zeros(N,1);
Vm = zeros(N,1);
Vm(GenInd)=[1.05 1.045 1.01 1.01 1.05 1.05]';   % Alsac's initial point
PgList(GenInd)=[0 80 50 20 20 20]'/MVA;
Taps = 1 ./ (1+[-2.2 -3.1 -6.8 -3.2]'/100);
bt = BusTypes; bt(GenInd)=3; bt(SlackInd)=1;
Qc=[.189; .04];    % Take Alsac literally: Q = V^2 / X = 1/X
QcList(QcInd) = Qc;
Pdata = PgList - Pd;
Qdata = QgList - Qd + QcList;

VmMin = VmLim(:,1); VmMax=VmLim(:,2); VmSpread=VmMax-VmMin;
PMin = PLim(:,1); PMax=PLim(:,2); PSpread=PMax-PMin;
QMin = QLim(:,1); QMax=QLim(:,2); QSpread=QMax-QMin;
QcMin = QcLim(:,1); QcMax=QcLim(:,2); QcSpread=QcMax-QcMin;
AMin = -pi/2; AMax = pi/2; ASpread = AMax-AMin;
Ptol = .01;   % Ignore P and Q errors less than Ptol

for ct=1:length(TInd)
  TapY(ct)=-Ybus(TInd(ct,1),TInd(ct,2))/OTaps(ct);
end

% If "Taps" is defined, change tap settings to the specified value
if exist('Taps')
  % Taps follow Debs' convention (1:t turns ratio).
  % Gross' turns ration is c:1.  Thus, c=1/t.
  for ct=1:length(TInd);
    P1 = TInd(ct,1); P2 = TInd(ct,2);
    Ot = OTaps(ct); t = Taps(ct);
    Ybus([P1 P2],[P1 P2]) = Ybus([P1 P2],[P1 P2]) + [t*t-Ot*Ot Ot-t; Ot-t 0]*TapY(ct);
  end
  OTaps = Taps;  % Ybus is now based on "Taps"
else
  Taps = OTaps;
end % if exist('Taps')

% This OPF is used with vars. defined before Y is adjusted.
[Vm,Delta, SlackP, SlackQ, Pflow,Qflow] = OPF(Ybus,NodeList,bt,Pdata,Qdata,Vm,SlackAng);
CostCoeff = [0 2.00 .00375; 0 1.75 .0175; 0 1.00 .0625; ...
          0 3.25 .00834; 0 3.00 .0250; 0 3.00 .0250];
```

```
CostCoeff(:,2)=CostCoeff(:,2)*MVA;  % Convert for use with p.u.
CostCoeff(:,3)=CostCoeff(:,3)*MVA^2;

NOxCoeff=zeros(length(GenInd),1);   % Since NOx is ignored, set the coeffs to zero.
NOxMax = inf;

if exist('Qc')
   [Delta,Vm,Pg,Qg,Qc,Taps,FitHist,Cost,NOx,Penalty,Pflow,Qflow] =
OPF_NXGA(Ybus,NodeList,BusTypes,SlackAng,OTaps,TapY,
VmLim,PLim,QLim,QcLim,TLim,NOxMax,LineMVA,
Pd,Qd,GenInd,QcInd,TInd,CostCoeff,NOxCoeff,SlackInd, Vm,Delta,Qc,Taps);
else
   [Delta,Vm,Pg,Qg,Qc,Taps,FitHist,Cost,NOx,Penalty,Pflow,Qflow] =
OPF_NXGA(Ybus,NodeList,BusTypes,SlackAng,OTaps,TapY,
VmLim,PLim,QLim,QcLim,TLim,NOxMax,LineMVA,
Pd,Qd,GenInd,QcInd,TInd,CostCoeff,NOxCoeff,SlackInd);
end
```

C.2 Init118.m (Initialize 118-bus system)

```
% Define the fundamental constraints and other constants as global variables
global N Nc NumGenU CostCoeff NOxCoeff GenInd QcInd AMin AMax
global VmMin VmMax PMin PMax QMin QMax QcMin QcMax NOxMax Ptol LineMVA

[Ybus,NodeList,BusTypes,Pd,Qd,Vg,SlackAng,LineMVA,Xform]=PwrData('118bus.dat');
MVA = 100; % MVA base for p.u.
% Don't limit the shunt power at each bus, so set its max to inf.
LineMVA = LineMVA + diag(inf*ones(length(NodeList),1));
LineMVA = LineMVA*5;    % Since the data file assumes that all lines are limited to
                        % 1.0 p.u., adjust the amount here.

% Specify the maximum total NOx
NOxMax = 35;

N = size(Ybus,1);
SlackInd=find(BusTypes==1);
Pd(1)=51.0; Qd(1)=27.0;    % For this system, there is a load at the slack bus (Bus 1).

% Change some Ps and Qs to match G. F. Reid's paper.
Pd(8)=28; Qd(8)=31.6;
Pd(24)=13; Pd(27)=71; Pd(31)=36; Qd(37)=0;
Pd(40)=66; Pd(42)=96; Qd(69)=60;

Pd=Pd/MVA; Qd=Qd/MVA;      % Convert load data (given in MW/MVars) to p.u.

% Define the generation limits (in p.u.), from G. F. Reid's paper
% GenInd list the bus numbers that have generation
% PLim and QLim are the limits at those buses.
GenInd = [1 10 12 25 26 49 59 61 65 66 80 89 100 103];
PLim=zeros(118,2); QLim=zeros(118,2);
PLim(GenInd,:) = [1 7; 1 5.5; .1 3.5; .5 3.5; 1 4.5; .5 3.5; .5 3; ...
            .5 3; .5 5; .5 5; .5 5.5; 1 8; .5 3.5; 0 2];

QLim(GenInd,:) = [-3 3; -1.47 2; -.35 1.2; -.47 1.4; -10 10; -.85 2.1; -.6 1.8; ...
            -1 3;  -.67 2; -.67 2;  -1.65 2.8; -2.1 3; -.5 1.55; -.6 .6];
```

```
QcInd = [ 4  6 15 18 19 24 27 31 32 34 36 40 42 46 54 55 56 62 69 70 72 73 74 ...
        76 77 85 87 90 91 92 99 104 105 107 110 111 112 113 116];
QcLim = [-3 3; -.6 .6; -.1 .3; -.6 .6; -.6 .6; -3 3; -3 3; -3 3; -.6 .6; -.6 .6; ...
        -.6 .6; -3 3; -3 3; -1 1; -3 3; -.6 .6; -.6 .6; -.2 .2; -.6 .6; -.6 .6; ...
        -1 1; -1 1; -.6 .6; -.6 .6; -.2 .7; -.6 .6; -1 10; -3 3; -1 1; -.6 .6; ...
        -1 1; -.6 .6; -.6 .6; -2 2; -.6 .6; -1 10; -1 10; -1 2; -10 10];

VmLim=ones(N,1)*[.9 1.1];

TInd = Xform(:,1:2);
TLim = ones(length(TInd),1)*[.9 1.1];
OTaps = Xform(:,3);
% OTaps = Old Taps.  These are the nominal taps values specified in the
%    data file.

VmMin = VmLim(:,1); VmMax=VmLim(:,2); VmSpread=VmMax-VmMin;
PMin = PLim(:,1); PMax=PLim(:,2); PSpread=PMax-PMin;
QMin = QLim(:,1); QMax=QLim(:,2); QSpread=QMax-QMin;
QcMin = QcLim(:,1); QcMax=QcLim(:,2); QcSpread=QcMax-QcMin;
AMin = -pi/2; AMax = pi/2; ASpread = AMax-AMin;
Ptol = .01;   % Ignore P and Q errors less than Ptol

for ct=1:length(TInd)
   TapY(ct)=-Ybus(TInd(ct,1),TInd(ct,2))/OTaps(ct);
end

if ~exist('Taps')
   Taps=1 ./ OTaps;     % Convert to Debs' notation for taps.
   Taps(8)=0.95;  % This seems to fit the Q data better.
end   % If NOT exist('Taps')

% If "Taps" is defined, change tap settings to the specified value
if exist('Taps')
% Taps follow Debs' convention (1:t turns ratio).
% Gross' turns ratio is c:1.  Thus, c=1/t.
for ct=1:length(TInd);
   P1 = TInd(ct,1); P2 = TInd(ct,2);
   Ot = OTaps(ct); t = Taps(ct);
   Ybus([P1 P2],[P1 P2]) = Ybus([P1 P2],[P1 P2]) + [t*t-Ot*Ot Ot-t; Ot-t 0]*TapY(ct);
end
OTaps = Taps;  % Ybus is now based on "Taps"

else
   Taps = OTaps;
end % if exist('Taps')

if ~exist('Qc')
   % Now use Start118.m to initialize the variables
   Start118;   % This defines Vm and Delta, but not Qc.  Taps is unchanged.
   PgList=zeros(N,1); QgList=zeros(N,1); QcList=zeros(N,1);
   [Fn_P, Fn_Q]=lf_eqs(Ybus, Delta, Vm, -1);
   Pg = Fn_P + Pd;
   Qg_plus_Qc = Fn_Q + Qd;
   PgList(GenInd) = Pg(GenInd);
   QgList(GenInd) = Qg_plus_Qc(GenInd);
```

```
  Qc = Qg_plus_Qc(QcInd);    % Qc is whatever Q is generated at buses with compensation.
  QcList(QcInd) = Qc;
  Pdata = PgList - Pd;
  Qdata = QgList + QcList - Qd;
  Vg = Vm; Vg([SlackInd GenInd])=[];
  Dg = Delta; Dg(SlackInd)=[];
  bt = BusTypes; bt(GenInd)=3; bt(SlackInd)=1;
  [Vm,Delta, SlackP, SlackQ, Pflow,Qflow] =
FDLF(Ybus,NodeList,bt,Pdata,Qdata,Vm,SlackAng,Dg,Vg);
  PgList(SlackInd)=SlackP+Pd(SlackInd);  QgList(SlackInd)=SlackQ+Qd(SlackInd);
end   % if exist('Qc')

% For convenience, G. F. Reid's cost function is scaled by 1E-3.
% Each row is in ascending order: Coeffs are for [x^0 x^1 x^2].
CostCoeff = [150 189 50; 115 200 55; 40 350 60; 122 315 55; 125 305 50; ...
          120 275 70; 70 345 70; 70 345 70; 130 245 50; 130 245 50; ...
          135 235 55; 200 160 45; 70 345 70; 45 389 60]/1000;

% NOx coeffs, ascending order.  The function gives lb NOx/hr as a fn. of the
%   generator's load (as a fraction of capacity: 1.00=full load).
%   In this case, almost all generators use the 320-MW curve for their percentages.
%   Only Bus 103 (Pmax = 200 MW) uses the 215-MW percentage curve.
% NOx = Pmax*polyval(fliplr(NOxCoeff(:,GenIndex)),LoadFraction);
NOxCoeff = ones(length(GenInd),1)*[0.1333 -0.2714 1.4460];  % Use the 320-MW curve
NOxCoeff(14,:) = [0.1816 -0.08205 1.5244];    % Replace Bus 103's curve with the 215-MW curve

% Since Matlab assumes coeffs are in descending order, flip the coeff matrices
CostCoeff = fliplr(CostCoeff);
NOxCoeff = fliplr(NOxCoeff);

if exist('Qc')

[Delta,Vm,Pg,Qg,Qc,Taps,FitHist,Cost,NOx,Penalty,Pflow,Qflow]=OPF_NXGA(Ybus,NodeList,Bu
sTypes,SlackAng,OTaps,TapY,
VmLim,PLim,QLim,QcLim,TLim,NOxMax,LineMVA,Pd,Qd,GenInd,QcInd,TInd,CostCoeff,NOxCo
eff,SlackInd, Vm,Delta,Qc,Taps);
else

[Delta,Vm,Pg,Qg,Qc,Taps,FitHist,Cost,NOx,Penalty,Pflow,Qflow]=OPF_NXGA(Ybus,NodeList,Bu
sTypes,SlackAng,OTaps,TapY,
VmLim,PLim,QLim,QcLim,TLim,NOxMax,LineMVA,Pd,Qd,GenInd,QcInd,TInd,CostCoeff,NOxCo
eff,SlackInd);
end
```

C.3 PwrData.m

```
function [Ybus,NodeList,BusTypes,Pg,Qg,Vg,SlackAng,LineMVA,Xform]=PwrData(FileNam)
% [Ybus,BusList,BusTypes, P,Q,V,SlackAng,LineMVA,Xform] = PwrData(FileName)
%    Extracts the information from a power system's data file.
%       If the Filename is omitted, the function propts the user for one.
%
%    Ybus is the system's bus admittance matrix.
%    BusList is a list of the bus numbers, in the order they are used in Ybus.
%    BusTypes is a list of numbers defining the type of the corresponding bus
%         in BusList: 1 = Slack,   2 = Load (PQ bus),   3 = Generation (PV bus)
```

```
%
%    P is the net injected real power at all buses, in the same order as BusList.
%    Q is the net injected reactive power at PQ buses, same order as in BusList.
%    V is the bus voltage for PV buses, in the same order as in BusList.
%    SlackAng is the angle of the slack bus, in radians.  It is usually 0.
%    LineMVA is a matrix.  LineMVA(N1,N2) = the MVA rating of the line between
%       nodes N1 and N2.  It is assumed that only one equivalent line is modeled
%       between each pair of buses.
%    Xform is the location and default value of tap-changing transformers
%       Xform = [Send-bus  Rec-bus  tap-value]
%
% Any line in the data file beginning with the string '% ' (percent followed by
%   space) is treated as a comment and ignored.  Blank lines are also ignored.
%
% Example data lines.  All lines are of the form DEVICE + NODES + PARMS.
%    NODES are integers representing the nodes.  They do not have to be
%       consecutive.
%    Parameters may be in any order and may be omitted (to accept a default).
%
%    BUS 1 GENER P -1.2 V 1.01
%    LINE 1 2 R 1 X 0.1 Y 0.01 MVA 1.5
%    XFORM 1 2 T 1.0 RL 1.0 XL 1.0 Ys 0.01
%    SHUNT 1 G 10 B 0.01

if nargin < 1
   FileNam=input('Enter the data file''s name: ','s');
end
[Fid, ErrMess]=fopen(FileNam,'rt');
if Fid == -1
   error(ErrMess)
end

Ybus=[];
NodeList=[];
SlackInd=[];  % Index of the slack bus (used to make sure there is only one).
Xform=[];
LineNum=0;
while ~feof(Fid)
   LineNum=LineNum+1;         % Keep track of which line of the file we are on.
   LineStr=num2str(LineNum);  % Convert to string for error messages
   Line=fgetl(Fid);
   [Device, Args]=strtok(Line);         % Get the first token in the data line.
   if isempty(Device), Device='%'; end  % Treat blank line as a comment line.
   switch lower(Device)
      case 'bus'
         [Node1,Rest]=strtok(Args);  % Get the node
         N1=str2num(Node1);
         if isempty(NodeList), NodeList=N1; end
         ss1=find(NodeList==N1);
         if isempty(ss1)
            NodeList=[NodeList N1];
            ss1=length(NodeList);   % ss1 is the row of Ybus corresponding to N1
         end

         [BusType, Rest]=strtok(Rest);  % Read the bus type
         switch upper(BusType)
```

```
  case {'SLACK', 'SWING'}
    Type=1;
  case {'LOAD', 'PQ'}
    Type=2;
  case {'GENER', 'PV'}
    Type=3;
  otherwise
    fclose(Fid);  % Close the data file.
    error(['Unknown BUS type ' BusType ' at Line ' LineStr]);
end

% Set default parameters for the bus.
G=0; B=0; P=0; Q=0; V=1; Angle=0;
% Parse the rest of the data line to set the actual parameters
[Parm, Rest]=strtok(Rest);
while ~isempty(Parm)
  [StrValue, Rest]=strtok(Rest);
  Value=str2num(StrValue);
  switch upper(Parm)
    case 'P'
      if Type == 1
        fclose(Fid);  % Close the data file.
        error(['Line ' LineStr ': P is not known at the slack bus'])
      end
      P=Value;
    case 'Q'
      if Type ~= 2
        fclose(Fid);  % Close the data file.
        error(['Line ' LineStr ': Q is only known at PQ buses'])
      end
      Q=Value;
    case 'V'
      if Type == 2
        fclose(Fid);  % Close the data file.
        error(['Line ' LineStr ': V is not known at PQ buses'])
      end
      V=Value;
    case 'ANGLE'
      if Type ~= 1
        fclose(Fid);  % Close the data file.
        error(['Line ' LineStr ': Angle is only known at slack bus'])
      end
      Angle=Value;
    case 'G'
      G=Value;
    case 'B'
      B=Value;
    otherwise
      fclose(Fid);  % Close the data file.
      error(['Unknown BUS Parm. ' Parm ' at Line ' LineStr]);
  end
  [Parm, Rest]=strtok(Rest);
end
if Type==1  % Make sure there is only one slack bus
  if isempty(SlackInd)
    SlackAng=Angle; % Define the slack bus angle.
```

```
      else
        disp('Two slack buses have been defined:')
        disp(NodeList([SlackInd ss1]))
        fclose(Fid);  % Close the data file.
        error('Only one bus can be the slack bus.');
      end
    end
    Pg(ss1)=P;
    Vg(ss1)=V;
    Qg(ss1)=Q;
    BusTypes(ss1)=Type;

    % Now add the bus' shunt to the Ybus matrix.
    if length(Ybus) < ss1, Ybus(ss1,ss1)=0; end
    Ybus(ss1,ss1)=Ybus(ss1,ss1) + G + i*B;

  case 'line'
    [Node1,ArgsN1]=strtok(Args);  % Get the two nodes
    [Node2,Rest]=strtok(ArgsN1);
    N1=str2num(Node1);
    if isempty(NodeList), NodeList=N1; end
    ss1=find(NodeList==N1);
    if isempty(ss1)
      NodeList=[NodeList N1];
      ss1=length(NodeList);   % ss1 is the row of Ybus corresponding to N1
    end
    N2=str2num(Node2);
    ss2=find(NodeList==N2);
    if isempty(ss2)
      NodeList=[NodeList N2];
      ss2=length(NodeList);   % ss1 is the row of Ybus corresponding to N2
    end

    % Set default parameters for the line.
    R=1; X=0.1; Y=0; MVA=inf;
    % Parse the rest of the data line to set the actual parameters
    [Parm, Rest]=strtok(Rest);
    while ~isempty(Parm)
      [StrValue, Rest]=strtok(Rest);
      Value=str2num(StrValue);
      switch upper(Parm)
        case 'R'
          R=Value;
        case 'X'
          X=Value;
        case {'Y','B'}
          Y=i*Value;
        case 'MVA'
          MVA=Value;
        otherwise
          fclose(Fid);  % Close the data file.
          error(['Unknown LINE Parm. ' Parm ' at Line ' LineStr]);
      end
      [Parm, Rest]=strtok(Rest);
    end
    % Now add the new line to the Ybus matrix.
```

```
   Yser = 1/(R+i*X);  % Series admittance of the line.
   MaxSS=max(ss1,ss2);
   if length(Ybus) < MaxSS
      Ybus(MaxSS,MaxSS)=0;
      LineMVA(MaxSS,MaxSS)=0;
   end
   Ybus(ss1,ss1)=Ybus(ss1,ss1) + Yser + .5*Y;
   Ybus(ss1,ss2)=Ybus(ss1,ss2) - Yser;
   Ybus(ss2,ss1)=Ybus(ss2,ss1) - Yser;
   Ybus(ss2,ss2)=Ybus(ss2,ss2) + Yser + .5*Y;
   % Define the line's MVA rating.
   LineMVA(ss1,ss2) = MVA;
   LineMVA(ss2,ss1) = MVA;

case {'xform','transformer'}
   [Node1,ArgsN1]=strtok(Args);  % Get the two nodes
   [Node2,Rest]=strtok(ArgsN1);
   N1=str2num(Node1);
   if isempty(NodeList), NodeList=N1; end
   ss1=find(NodeList==N1);
   if isempty(ss1)
      NodeList=[NodeList N1];
      ss1=length(NodeList);   % ss1 is the row of Ybus corresponding to N1
   end
   N2=str2num(Node2);
   ss2=find(NodeList==N2);
   if isempty(ss2)
      NodeList=[NodeList N2];
      ss2=length(NodeList);   % ss1 is the row of Ybus corresponding to N2
   end

   % Set default parameters for the transformer.
   t=1; RL=.1; XL=0; Ys=0; MVA=inf;
   % Parse the rest of the data line to set the actual parameters
   [Parm, Rest]=strtok(Rest);
   while ~isempty(Parm)
      [StrValue, Rest]=strtok(Rest);
      Value=str2num(StrValue);
      switch upper(Parm)
         case {'RL','R'}
            RL=Value;
         case {'XL','X'}
            XL=Value;
         case {'YS','B'}
            Ys=i*Value;
         case 'T'
            t=Value;
         case 'MVA'
            MVA=Value;
         otherwise
            fclose(Fid);  % Close the data file.
            error(['Unknown TRANSFORMER Parm. ' Parm ' at Line ' LineStr]);
      end
      [Parm, Rest]=strtok(Rest);
   end
   % Now add the new transformer to the Ybus matrix.
```

```
   YL = 1/(RL+i*XL);
   Y12 = t*YL;
   Ys1 = t*(t-1)*YL + t*t*Ys;
   Ys2 = (1-t)*YL;
   MaxSS=max(ss1,ss2);
   if length(Ybus) < MaxSS
      Ybus(MaxSS,MaxSS)=0;
      LineMVA(MaxSS,MaxSS)=0;
   end
   Ybus(ss1,ss1)=Ybus(ss1,ss1) + Y12 + Ys1;
   Ybus(ss1,ss2)=Ybus(ss1,ss2) - Y12;
   Ybus(ss2,ss1)=Ybus(ss2,ss1) - Y12;
   Ybus(ss2,ss2)=Ybus(ss2,ss2) + Y12 + Ys2;
   % Define the transformer's MVA rating.
   LineMVA(ss1,ss2) = MVA;
   LineMVA(ss2,ss1) = MVA;
   % Place transformer data in Xform matrix
   % Xform = [Send-bus  Rec-bus  tap-value
   Xform=[Xform; ss1 ss2 t];

case 'shunt'
   [Node1,Rest]=strtok(Args);  % Get the node
   N1=str2num(Node1);
   if isempty(NodeList), NodeList=N1; end
   ss1=find(NodeList==N1);
   if isempty(ss1)
      NodeList=[NodeList N1];
      ss1=length(NodeList);   % ss1 is the row of Ybus corresponding to N1
   end

   % Set default parameters for the shunt device.
   G=0; B=0;
   % Parse the rest of the data line to set the actual parameters
   [Parm, Rest]=strtok(Rest);
   while ~isempty(Parm)
      [StrValue, Rest]=strtok(Rest);
      Value=str2num(StrValue);
      switch upper(Parm)
         case 'G'
            G=Value;
         case 'B'
            B=Value;
         otherwise
            fclose(Fid);  % Close the data file.
            error(['Unknown SHUNT Parm. ' Parm ' at Line ' LineStr]);
      end
      [Parm, Rest]=strtok(Rest);
   end
   % Now add the shunt to the Ybus matrix.
   if length(Ybus) < ss1, Ybus(ss1,ss1)=0; end
   Ybus(ss1,ss1)=Ybus(ss1,ss1) + G + i*B;

case '%'
   % Ignore a blank line or a comment.

otherwise
```

```
        fclose(Fid);  % Close the data file.
        error(['Unknown Device ' Device ' at Line ' LineStr]);
    end
end

fclose(Fid);  % Close the data file.
% Convert Pg, Qg, Vg to column vectors
Pg=Pg'; Qg=Qg'; Vg=Vg';
```

C.4 Start118.m

```
% Starting point for IEEE 118-bus system, as provided in G.F. Reid's paper
% The angles are given in degrees.  They are converted to radians at the end of
%    this file.
% SlackInd must contain the index of the slack bus (1 in this case).

% x0 = [|V|, Delta]
x0=[1.035 0; .971 -7.1; .968 -7.41; .998 -4.64; 1.002 -4.31; .99 -6.06; ...
   .989 -6.12; 1.015 -.76; 1.046 5.19; 1.05 11.44; .985 -6.14; .99 -5.85; ...
   .968 -8.12; .983 -7.47; .97 -10.3; .984 -7.32; .995 -8; .973 -10.27; ...
   .962 -10.79; .958 -10.2; .959 -8.82; .97 -6.51; 1 -1.97; .992 -3.2; ...
   1.05 6.02; 1.015 7.52; .968 -6.81; .962 -8.5; .9 -9.46; .986 -3.44; ...
   .967 -9.33; .963 -7.45; .971 -11.88; .984 -12.29; .981 -12.74; .98 -12.73; ...
   .991 -11.84; .963 -6.87; .97 -15.54; .97 -16.79; .96 -17.37; .985 -16.15; ...
   .978 -12.9; .986 -11.27; .987 -9.75; 1.005 -7.29; 1.018 -5.5; 1.021 -5.7; ...
   1.025 -4.66; 1.001 -6.6; .972 -9.17; .961 -10.07; .948 -10.91; .955 -9.91; ...
   .952 -10.17; .954 -10; .971 -8.94; .962 -9.82; .985 -5.45; .993 -2.21; ...
   .995 -1.32; .998 -2.14; .97 -2.78; .985 -1.4; 1.005 .53; 1.05 .95; ...
   1.02 -1.26; 1.005 -.85; .955 -8.10; .984 -6.17; .987 -6.2; .98 -5.34; ...
   .991 -6.41; .958 -7.9; .967 -6.89; .943 -8.56; 1.006 -4.37; 1.003 -4.71; ...
   1.009 -4.48; 1.04 -2.41; .997 -1.42; .99 -4.87; .986 -4.34; .981 -2.86; ...
   .985 -1.82; .987 -3.19; 1.015 -2.93; .988 .41; 1.005 3.85; .985 -2.12; ...
   .98 -1.55; .99 -.16; .987 -1.99; .991 -3.12; .981 -4.11; .993 -4.29; ...
   1.012 -3.7; 1.024 -3.52; 1.01 -3.46; 1.017 -2.12; .993 -2.16; .99 -.93; ...
   1.01 -2; .971 -6.04; .965 -7.03; .962 -7.74; .952 -10.31; .966 -7.95; ...
   .967 -8.28; .973 -8.82; .98 -7.18; .975 -11.92; .993 -8.07; .96 -7.74; ...
   .96 -7.75; 1.005 -1.27; .974 -7.4; .949 -8.13];

Vm = x0(:,1);
Delta = (x0(:,2)-x0(SlackInd,2))*pi/180;
```

C.5 OPF_NXGA.m

```
function [Delta,Vm,Pg,Qg,Qc,Taps,PFitHist,Cost,NOx,Penalty,Pflow,Qflow] =
OPF_NXGA(Ybus,NodeList,BusTypes,SlackAng,Taps0,TapY, VmLim, PLim, QLim,
QcLim,TLim,NOxMax,LineMVA, Pd, Qd, GenInd,QcInd,TInd, CostCoeff,NOxCoeff, SlackInd,
Vm,Delta,Qc,Taps)

% Define the fundamental constraints and other constants as global variables
global N Nc NumGenU CostCoeff NOxCoeff GenInd QcInd AMin AMax
global VmMin VmMax PMin PMax QMin QMax QcMin QcMax NOxMax Ptol LineMVA

VmSpread=VmMax-VmMin;
```

```
PSpread=PMax-PMin;
QSpread=QMax-QMin;
QcSpread=QcMax-QcMin;
TMin = TLim(:,1); TMax=TLim(:,2); TSpread=TMax-TMin;
ASpread = AMax-AMin;

PGens= 10;     % Number of GA generations
PPopSize=20;   % Number of solutions that make up a GA population
PNumRep=round(max(0.25*PPopSize, 2));  % Number bad solutions to replace
PNumElite=round(max(0.2*PPopSize, 2)); % Of PNumRep, number to replace via elitism
PNumRnd=0;

PUniformMut=.01;    % Prob. of uniform parameter mutation (Michalewicz page 111)
PNonUniMut =.01;    % Prob. of non-uniform parm. mutation
b=2;      % Factor of how fast non-uniform mutation becomes local (Mich. page 112)
PMutPowProb=0;    % Prob. of multiplying a parm. by a power of 10
PAXProb=.02;   % Prob. arithmetic crossover
PSXProb=.02;   % Prob. simple crossover
% Re-set APop to be coeffs of nullspace
NA2v = .02;  % Scale factor to convert from NAPop to v

N = size(VmLim,1);  % N = number of buses.
Ng = length(GenInd);
Nc = length(QcInd);
Nt = length(TInd);
Na = 2*Ng;
PFitHist=zeros(PGens+1,1);
SlackGen = find(GenInd==SlackInd);

NAPop = AMin + ASpread*rand(PPopSize,Na);
OAPop = zeros(size(NAPop));
QcMinPop = ones(PPopSize,1)*QcMin';
QcSprPop = ones(PPopSize,1)*QcSpread';
NQcPop = QcMinPop + rand(PPopSize,Nc).*QcSprPop;
TMinPop = ones(PPopSize,1)*TMin';
TSprPop = ones(PPopSize,1)*TSpread';
NTPop = TMinPop + rand(PPopSize,Nt).*TSprPop;

bt = BusTypes; bt(GenInd)=3; bt(SlackInd)=1;
LoadInd = ones(1,N);
LoadInd(GenInd)=0; LoadInd(SlackInd)=0;
% Load buses (where P or Q is fixed)
% Note that Q is NOT fixed at any bus having Qc or a tap-changing transformer.
%   Thus, those buses must be removed from QLoadInd
PLoadInd=find(LoadInd);
QLoadInd=LoadInd;
QLoadInd=find(QLoadInd);
lss = [PLoadInd QLoadInd+N]; % Load subscripts of Ybus
QQcInd = [GenInd QcInd];
QcDV = 1:2*N;

LF_Vm = zeros(PPopSize,N);    % Where we store an x found by LF
LF_D = zeros(PPopSize,N);
OLF_VM = LF_Vm;
OLF_D = LF_D;
```

```
NotSlackA=ones(1,2*N);     % Not slack angle (but does include Vslack)
NotSlackA(SlackInd)=0;
NotSlackA=find(NotSlackA);

Tss = zeros(1,N); Tss(TInd(:))=1; Tss=find(Tss); Tss=[Tss Tss+N];
if nargin > 20    % Seed the initial pop. with 2 copies of the specified guess
  NQcPop=ones(PPopSize,1)*Qc' + .01*NQcPop;
  NTPop=ones(PPopSize,1)*Taps' + .005*NTPop;
  NQcPop(1:2,:)=[1;1]*Qc';
  NTPop(1:2,:)=[1;1]*Taps';
  NAPop(1:2,:)=0;

  % Make sure Taps are the same as what's assumed by Ybus.
  for ct=1:length(TInd);
    P1 = TInd(ct,1); P2 = TInd(ct,2);
    Ot = Taps0(ct); t = Taps(ct);
    Ybus([P1 P2],[P1 P2]) = Ybus([P1 P2],[P1 P2]) + [t*t-Ot*Ot Ot-t; Ot-t 0]*TapY(ct);
  end
  Taps0 = Taps;  % Ybus is now based on "Taps"

else
  % Do an ordinary load flow to set the initial guess
  PgList=zeros(N,1);QgList=zeros(N,1);QcList=zeros(N,1);
  PgList(GenInd)=1.05*PMax(GenInd)*sum(Pd)/sum(PMax);
  QgList(GenInd)=1.05*QMax(GenInd)*sum(Qd)/sum(QMax);
  Taps=Taps0;
  Qc = .5*QcMax;
  QcList(QcInd) = Qc;
  Pdata = PgList - Pd;
  Qdata = QgList - Qd + QcList;
  Vm = .95*ones(N,1); Vm(GenInd)=1.05; Vm(SlackInd)=1.05;
  GuessVm=.95*ones(N-Ng,1);
  GuessDelta = -5*pi/180*ones(N-1,1);
  [Vm,Delta, SlackP, SlackQ, Pflow,Qflow] =
FDLF(Ybus,NodeList,bt,Pdata,Qdata,Vm,SlackAng,GuessDelta,GuessVm);
  NQcPop(1:2,:)=[1;1]*Qc';
  NTPop(1:2,:)=[1;1]*Taps';
  NAPop(1:2,:)=0;
end

OYbus = Ybus;
Vm0=Vm; Delta0=Delta; Qc0=Qc;

% Taps0 is specified in the arguments.
J0=OPF_Jacb(Ybus, Delta0, Vm0);
Jr=J0(lss,:);
nJ0 = null(Jr);        % nJ0 is the reference nullspace

QcList=zeros(N,1);
VmMinRnd = ones(PNumRnd,1)*VmMin';
VmSprRnd = ones(PNumRnd,1)*VmSpread';
QcMinRnd = QcMinPop(1:PNumRnd,:);
QcSprRnd = QcSprPop(1:PNumRnd,:);
TMinRnd = TMinPop(1:PNumRnd,:);
TSprRnd = TSprPop(1:PNumRnd,:);
NumGenU = length(GenInd);
```

```
PFit=zeros(PPopSize,1);
PCost=zeros(PPopSize,1);
PPen=zeros(PPopSize,1);
PNOx=zeros(PPopSize,1);
POldFit=zeros(PPopSize,1);
PBestOldFit=0;

Pwt = 1000; % Penalty weight
for Pgen=0:PGens
  LF_Used=zeros(PPopSize,1);    % Flag that says whether a Load-flow was used
                        % instead of the nullspace vector.
  for Ptry=1:PPopSize
    Qc = NQcPop(Ptry,:)';
    Taps = NTPop(Ptry,:)';
    % Quantize Taps and Qc by rounding to the nearest 0.01
    Taps = round(100*Taps)/100;
    Qc = round(100*Qc)/100;
    NQcPop(Ptry,:) = Qc';
    NTPop(Ptry,:) = Taps';

    v = NAPop(Ptry,:)'*NA2v;
    QcList(QcInd)=Qc;

    % Adjust Ybus & J to account for tap settings, if they've changed.
    % The turns ratio is 1:t and follows Debs (and is 1/c in Gross)
    if ~all(Taps==Taps0)
      dx = nJ0*v;    % To start, find x using the old nullspace.
      Delta = Delta0 + dx(1:N) - dx(SlackInd);
      Vm = Vm0 + dx(N+1:2*N);
      [Ybus,J2] = J_NewTaps(Delta,Vm,OYbus,J0,Taps0,Taps,TInd,TapY);
      Jr2 = J2(lss,:); nJ2 = null(Jr2);
      v = NAPop(Ptry,:)'*NA2v;
      dx = nJ2*nJ2'*(nJ0*v);
    else
      Ybus = OYbus;
      J2=J0; nJ2=nJ0;
      v = NAPop(Ptry,:)'*NA2v;
      dx = nJ0*v;
      dSt = zeros(2*N,1);
    end   % If taps have changed

    dxQc = nJ2*nJ2(QcDV,:)'*(J2(QcInd+N,QcDV)\(Qc-Qc0));
    Delta = Delta0 + dx(1:N) - dx(SlackInd)+ dxQc(1:N) - dxQc(SlackInd);
    Vm = Vm0 + dx(N+1:2*N) + dxQc(N+1:2*N);

    [Cost, NOx, Penalty, Pflow, Qflow, NQcList] = fitness(Delta,Vm,QcList,Ybus,Pd,Qd);
    NQcPop(Ptry,:) = NQcList(QcInd)';
    PFit(Ptry)=1/(1+Cost+Pwt*Penalty);
    PCost(Ptry)=Cost;
    PPen(Ptry)=Penalty;
    PNOx(Ptry)=NOx;
    disp(['Gen-Try-Cost-Penalty-Fit     ' num2str(Pgen) '  ' num2str(Ptry) '  ' num2str(Cost) '  '
num2str(Penalty) '  ' num2str(1/(1+Cost+Pwt*Penalty))])
```

```
  if (PFit(Ptry) > .2*PBestOldFit) & (PPen(Ptry) > .01) & PPen(Ptry)<10
    % Run a load-flow to make sure that the load bus P's and Q's are right.
    % First, calculate P and Q resulting from the states
    [Fn_P, Fn_Q]=lf_eqs(Ybus, Delta, Vm, -1);
    Pg = Fn_P + Pd;
    Qg = Fn_Q + Qd - QcList;

    % Now, force P and Q to be zero at the load buses.
    PgList=zeros(N,1);  QgList=zeros(N,1);  QcList=zeros(N,1);
    PgList(GenInd)=Pg(GenInd); QgList(GenInd)=Qg(GenInd);
    QcList(QcInd) = Qc;
    Pdata = PgList - Pd;
    Qdata = QgList + QcList - Qd;

    Delta1 = Delta0 + dx(1:N) - dx(SlackInd);
    Vm1 = Vm0 + dx(N+1:2*N);
    Vg = Vm1; Vg(bt~=2)=[];
    Dg = Delta1; Dg(SlackInd)=[];
    [Vm,Delta, SlackP, SlackQ, Pflow,Qflow] =
FDLF(Ybus,NodeList,bt,Pdata,Qdata,Vm1,SlackAng,Dg,Vg);
    [Cost, NOx, Penalty, Pflow, Qflow, NQcList] = fitness(Delta,Vm,QcList,Ybus,Pd,Qd);
    NQcPop(Ptry,:) = NQcList(QcInd)';
    PFit(Ptry)=1/(1+Cost+Pwt*Penalty);
    PCost(Ptry)=Cost;
    PPen(Ptry)=Penalty;
    PNOx(Ptry)=NOx;
    J3=OPF_Jacb(Ybus, Delta, Vm);
    nJ3 = null(J3);
    dx3 = [Delta-Delta0; Vm-Vm0];
    dx3 = dx3 + nJ3*nJ3(QcDV,:)'*(J3(QcInd+N,QcDV)\(Qc0-Qc));
    v3 = dx3'*nJ0;     % Actually v'
    NAPop(Ptry,:)=v3/NA2v;
    LF_Vm(Ptry,:)=Vm';
    LF_D (Ptry,:)=Delta';
    LF_Used(Ptry)=1;
    disp(['        Updated Cost-Penalty-Fit '  num2str(Cost) '  ' num2str(Penalty) '  '
num2str(1/(1+Cost+Pwt*Penalty))])
  end

  end

  [PSortFit, PFitInd]=sort(PFit);
  % ELITISM: Replace the worst new genes with good old ones (some are chosen randomly)
  if Pgen > 0
    BadNew = PFitInd(1:PNumRep);
    GoodOld = [POldFitInd(PPopSize-PNumElite+1:PPopSize); zeros(PNumRep-PNumElite,1)];
    OldCumFit = cumsum(POldFit)/sum(POldFit);
    for NewGene=1:PNumRep-PNumElite   % Roulette wheel for Old pop.
      GoodOld(PNumElite+NewGene)=min(find(OldCumFit>=rand));
    end
    NAPop(BadNew,:) = OAPop(GoodOld,:);
    NQcPop(BadNew,:) = OQcPop(GoodOld,:);
    NTPop(BadNew,:) = OTPop(GoodOld,:);
    PFit(BadNew) = 1 ./ (1+POCost(GoodOld)+Pwt*POPen(GoodOld));
    PCost(BadNew) = POCost(GoodOld);
    PPen(BadNew) = POPen(GoodOld);
```

```
      LF_Used(BadNew) = OLF_Used(GoodOld);
      LF_Vm(BadNew,:) = OLF_Vm(GoodOld,:);
      LF_D(BadNew,:) = OLF_D(GoodOld,:);
    end
    PBestFit=max(PFit);
    disp(['Generation ' num2str(Pgen) ' Max Fitness = ' num2str(PBestFit)])
    PFitHist(Pgen+1)=PBestFit;
    save endgen

    if PBestFit == 0, break, end  % If all fitnesses = 0, give up on this gene.
                                  % It's impossible to improve this model.

    if (Pgen < PGens) & (PBestFit > 0)
   % If not the last generation & pop. not extinct, produce the next generations

 if PBestFit > 1.01*PBestOldFit
    % If the fitness has been improved by at least 1%, re-calculate
    %    the Jacobian and the null space.
    % The old perturbations must be projected onto the new null space.
    % The reference state is adjusted to reflect the best solution so far.
    PBestOldFit=PBestFit;
    bss=min(find(PFit==max(PFit)));
    Taps = NTPop(bss,:)';
    Qc = NQcPop(bss,:)';
    v = NAPop(bss,:)'*NA2v;

    % Quantize Taps and Qc by rounding to the nearest 0.01
    Taps = round(100*Taps)/100;
    Qc = round(100*Qc)/100;
    QcList(QcInd)=Qc;

    if ~LF_Used(bss)
      % Adjust Ybus & J to account for tap settings, if they've changed.
      % The turns ratio is 1:t and follows Debs (and is 1/c in Gross)
      if ~all(Taps==Taps0)
        dx = nJ0*v;    % To start, find x using the old nullspace.
        Delta = Delta0 + dx(1:N) - dx(SlackInd);
        Vm = Vm0 + dx(N+1:2*N);
        [Ybus,J2] = J_NewTaps(Delta,Vm,OYbus,J0,Taps0,Taps,TInd,TapY);
        Jr2 = J2(lss,:); nJ2 = null(Jr2);
        dx = nJ2*nJ2'*(nJ0*v);
      else
        Ybus = OYbus;
        J2=J0; nJ2=nJ0;
        dx = nJ0*v;
      end   % If taps have changed
      dx = dx + nJ2*nJ2(QcDV,:)'*(J2(QcInd+N,QcDV)\(Qc-Qc0));
      Delta = Delta0 + dx(1:N) - dx(SlackInd);
      Vm = Vm0 + dx(N+1:2*N);

    else % if ~LF_Used
      Delta = LF_D(bss,:)';
      Vm = LF_Vm(bss,:)';
      if ~all(Taps==Taps0)
        [Ybus,J2] = J_NewTaps(Delta,Vm,OYbus,J0,Taps0,Taps,TInd,TapY);
        Jr2 = J2(lss,:); nJ2 = null(Jr2);
```

```
    else
      Ybus = OYbus;
      J2=J0; nJ2=nJ0;
    end   % If taps have changed
  end % if ~LF_Used

  % Run a load-flow to make sure that the load bus P's and Q's are right.
  % First, calculate P and Q resulting from the states
  [Fn_P, Fn_Q]=lf_eqs(Ybus, Delta, Vm, -1);
  Pg = Fn_P + Pd;
  Qg = Fn_Q + Qd - QcList;

  % Now, force P and Q to be zero at the load buses.
  PgList=zeros(N,1);  QgList=zeros(N,1);  QcList=zeros(N,1);
  PgList(GenInd)=Pg(GenInd); QgList(GenInd)=Qg(GenInd);
  QcList(QcInd) = Qc;
  Pdata = PgList - Pd;
  Qdata = QgList + QcList - Qd;
  Vg = Vm; Vg(bt~=2)=[];
  Dg = Delta; Dg(SlackInd)=[];

  % Figure out what the new reference will be, but don't update yet.
  [NVm0,NDelta0, SlackP, SlackQ, Pflow,Qflow] =
FDLF(Ybus,NodeList,bt,Pdata,Qdata,Vm,SlackAng,Dg,Vg);
  PgList(SlackInd)=SlackP+Pd(SlackInd);  QgList(SlackInd)=SlackQ+Qd(SlackInd);
  NOYbus = Ybus;
  NJ0=OPF_Jacb(NOYbus, NDelta0, NVm0);
  NJr=NJ0(lss,:);
  NTaps0 = Taps; NQc0 = Qc;
  NnJ0 = null(NJr);

  % Now find the new "v" vectors for all members of the population
  for ct = 1:PPopSize
    Qc = NQcPop(ct,:)';
    Taps = NTPop(ct,:)';
    % Quantize Taps and Qc by rounding to the nearest 0.01
    Taps = round(100*Taps)/100;
    Qc = round(100*Qc)/100;
    NQcPop(ct,:) = Qc';
    NTPop(ct,:) = Taps';

    v = NAPop(ct,:)'*NA2v;
    QcList(QcInd)=Qc;

  if ~LF_Used(ct)
    % Adjust Ybus & J to account for tap settings, if they've changed.
    % The turns ratio is 1:t and follows Debs (and is 1/c in Gross)
    if ~all(Taps==Taps0)
      dx = nJ0*v;    % To start, find x using the old nullspace.
      Delta = Delta0 + dx(1:N) - dx(SlackInd);
      Vm = Vm0 + dx(N+1:2*N);
      [Ybus,J2] = J_NewTaps(Delta,Vm,OYbus,J0,Taps0,Taps,TInd,TapY);
      Jr2 = J2(lss,:); nJ2 = null(Jr2);
      dx = nJ2*nJ2'*(nJ0*v);
    else
      Ybus = OYbus;
```

```
      J2=J0; nJ2=nJ0;
      dx = nJ0*v;
    end   % If taps have changed

    % Now, Delta and Vm have been found under the OLD Ybus and Jacobian
    Delta = Delta0 + dx(1:N) - dx(SlackInd);
    Vm = Vm0 + dx(N+1:2*N);
    dx = dx + nJ2*nJ2(QcDV,:)'*(J2(QcInd+N,QcDV)\(Qc-Qc0));
    % Now, Delta and Vm have been found under the OLD Ybus and Jacobian
    Delta = Delta0 + dx(1:N) - dx(SlackInd);
    Vm = Vm0 + dx(N+1:2*N);

  else  % if ~LF_Used
    Delta = LF_D(ct,:)';
    Vm = LF_Vm(ct,:)';
    dx = zeros(2*N,1);

    if ~all(Taps==Taps0)
      [Ybus,J2] = J_NewTaps(Delta,Vm,OYbus,J0,Taps0,Taps,TInd,TapY);
      Jr2 = J2(lss,:); nJ2 = null(Jr2);
    else
      Ybus = OYbus;
      J2=J0; nJ2=nJ0;
    end   % If taps have changed

    J2=OPF_Jacb(Ybus, Delta, Vm);
    Jr2 = J2(lss,:); nJ2 = null(Jr2);
  end % if ~LF_Used


  % Now find the updated v, projected onto the new nullspace.  NAPop = v/NA2v
  % If the taps are not set to the new reference value, the algorithm will
  %   change nJ.  Thus, we must project v onto NnJ0, the nullspace based
  %   on the new REFERENCE taps, not the taps for this particular solution.

  J3 = J2; nJ3 = nJ2;
  dx3 = [Delta-NDelta0; Vm-NVm0];
  dx3 = dx3 + nJ3*nJ3(QcDV,:)'*(J3(QcInd+N,QcDV)\(NQc0-Qc));
  v3 = dx3'*NnJ0;     % Actually v'
  NAPop(ct,:) = v3/NA2v;
  [Cost, NOx, Penalty, Pflow, Qflow, NQcList] = fitness(Delta,Vm,QcList,Ybus,Pd,Qd);
  NQcPop(Ptry,:) = NQcList(QcInd)';
  PFit(ct)=1/(1+Cost+Pwt*Penalty);
  PCost(ct)=Cost;
  PPen(ct)=Penalty;
end

  % Update the reference state (Vm0, Delta0, etc.)
  Vm0 = NVm0;
  Delta0 = NDelta0;
  OYbus = NOYbus;
  J0 = NJ0;
  Jr = NJr;
  Taps0 = NTaps0; Qc0 = NQc0;
  nJ0 = NnJ0;
```

```
   % If the load-flow solution had to correct a load imbalance (because of
   %    the linearization or because of an imperfect initial guess), the reference
   %    entry's "v" vector won't be zero.
   % Since the difference between the reference case and itself (i.e., the reference
   %    "v") MUST be 0, we must subtract the reference "v" from every member
   %    of the "v" population.
   NQcPop(bss,:)=NQc0';
   NAPop(bss,:)=0;

end   % of the code that updates the reference state

      [PSortFit, PFitInd]=sort(PFit);
      POldFitInd=PFitInd;    % Keep track of the best old genes.
      POldFit=PFit;
      POCost=PCost;
      POPen =PPen;
      OAPop=NAPop;
      OQcPop=NQcPop;
      OTPop=NTPop;
      OLF_Vm = LF_Vm;
      OLF_D = LF_D;
      OLF_Used = LF_Used;

      % Selection:  Clone good genes, kill bad ones
      PFit=PFit/sum(PFit);
      PCumFit=cumsum(PFit);
      GoodNew = PFitInd(PPopSize-PNumElite+1:PPopSize);
      NAPop(1:PNumElite,:) = OAPop(GoodNew,:);  % Copy elites first
      NQcPop(1:PNumElite,:) = OQcPop(GoodNew,:);
      NTPop(1:PNumElite,:) = OTPop(GoodNew,:);

      % Add some random members
      NAPop(PNumElite+1:PNumElite+PNumRnd,:) = AMin + ASpread*rand(PNumRnd,Na);
      NQcPop(PNumElite+1:PNumElite+PNumRnd,:) = QcMinRnd +
rand(PNumRnd,Nc).*QcSprRnd;
      NTPop(PNumElite+1:PNumElite+PNumRnd,:) = TMinRnd + rand(PNumRnd,Nt).*TSprRnd;

      % Roulette wheel to fill out the new pop.
      for NewGene=PNumElite+PNumRnd+1:PPopSize
        choice=min(find(PCumFit>=rand));
        NAPop(NewGene,:) = OAPop(choice,:);
        NQcPop(NewGene,:) = OQcPop(choice,:);
        NTPop(NewGene,:) = OTPop(choice,:);
      end

      % Arithmetic Crossover, NOT applied to the whole vector
      for recomb=1:round(PAXProb*PPopSize/2)
        % Crossover for nullspace
        I1=floor(rand*PPopSize)+1;
        I2=floor(rand*PPopSize)+1;
        P1=NAPop(I1,:);
        P2=NAPop(I2,:);
        xf=rand;   %  Crossover Factor
        C1=xf*P1 + (1-xf)*P2;
        C2=(1-xf)*P1 + xf*P2;
        Xpos=find((rand(1,Na)) > 0.5*Pgen/PGens);  % Crossover positions
```

```
  NAPop(I1,Xpos)=C1(Xpos);   %  Replace parents with children
  NAPop(I2,Xpos)=C2(Xpos);   %    but only at positions indicated by Xpos
end

for recomb=1:round(PAXProb*PPopSize/2)
  % Crossover for Qc
  I1=floor(rand*PPopSize)+1;
  I2=floor(rand*PPopSize)+1;
  P1=NQcPop(I1,:);
  P2=NQcPop(I2,:);
  xf=rand;   %  Crossover Factor
  C1=xf*P1 + (1-xf)*P2;
  C2=(1-xf)*P1 + xf*P2;
  Xpos=find((rand(1,Nc)) > 0.5*Pgen/PGens);  % Crossover positions
  NQcPop(I1,Xpos)=C1(Xpos);   %  Replace parents with children
  NQcPop(I2,Xpos)=C2(Xpos);   %    but only at positions indicated by Xpos

  % Crossover for T
  I1=floor(rand*PPopSize)+1;
  I2=floor(rand*PPopSize)+1;
  P1=NTPop(I1,:);
  P2=NTPop(I2,:);
  xf=rand;   %  Crossover Factor
  C1=xf*P1 + (1-xf)*P2;
  C2=(1-xf)*P1 + xf*P2;
  Xpos=find((rand(1,Nt)) > 0.5*Pgen/PGens);  % Crossover positions
  NTPop(I1,Xpos)=C1(Xpos);   %  Replace parents with children
  NTPop(I2,Xpos)=C2(Xpos);   %    but only at positions indicated by Xpos
end

% Recombinitaton (Simple Crossover)
for recomb=1:round(PSXProb*PPopSize/2)
  % Crossover for nullspace
  I1=floor(rand*PPopSize)+1;
  I2=floor(rand*PPopSize)+1;
  P1=NAPop(I1,:);
  P2=NAPop(I2,:);
  % Two-point crossover
  Pos1=floor((Na-1)*rand)+1;   % Rand integer in [1, Na-1]
  Pos2=floor((Na-Pos1-2)*rand)+Pos1+1;   % Rand integer in [Pos1+1, Na-1]
  NAPop(I1,:)=[P1(1:Pos1) P2((Pos1+1):Pos2) P1(Pos2+1:Na)];
  NAPop(I2,:)=[P2(1:Pos1) P1((Pos1+1):Pos2) P2(Pos2+1:Na)];
end

for recomb=1:round(PSXProb*PPopSize/2)
 % Crossover for Qc
  I1=floor(rand*PPopSize)+1;
  I2=floor(rand*PPopSize)+1;
  P1=NQcPop(I1,:);
  P2=NQcPop(I2,:);
  % Two-point crossover
  Pos1=floor((Nc-1)*rand)+1;   % Rand integer in [1, Nc-1]
  Pos2=floor((Nc-Pos1-2)*rand)+Pos1+1;   % Rand integer in [Pos1+1, Nc-1]
  NQcPop(I1,:)=[P1(1:Pos1) P2((Pos1+1):Pos2) P1(Pos2+1:Nc)];
  NQcPop(I2,:)=[P2(1:Pos1) P1((Pos1+1):Pos2) P2(Pos2+1:Nc)];
```

```
  % Crossover for T
   I1=floor(rand*PPopSize)+1;
   I2=floor(rand*PPopSize)+1;
   P1=NTPop(I1,:);
   P2=NTPop(I2,:);
   % Two-point crossover
   Pos1=floor((Nt-1)*rand)+1;   % Rand integer in [1, Nt-1]
   Pos2=floor((Nt-Pos1-2)*rand)+Pos1+1;   % Rand integer in [Pos1+1, Nt-1]
   NTPop(I1,:)=[P1(1:Pos1) P2((Pos1+1):Pos2) P1(Pos2+1:Nt)];
   NTPop(I2,:)=[P2(1:Pos1) P1((Pos1+1):Pos2) P2(Pos2+1:Nt)];
end

% Uniform Parameter Mutation
for MutCt=1:round(PUniformMut*N*PPopSize);
   MutGene=floor(rand*PPopSize)+1;
   MutLoc=floor(rand*Na)+1;  % Rand.# between 1 and Na.
   NAPop(MutGene,MutLoc) = AMin + ASpread*rand;
end

for MutCt=1:round(PUniformMut*N*PPopSize);
   MutGene=floor(rand*PPopSize)+1;
   MutLoc=floor(rand*Nc)+1;  % Rand.# between 1 and Nc.
   NQcPop(MutGene,MutLoc) = QcMin(MutLoc) + QcSpread(MutLoc)*rand;

   MutGene=floor(rand*PPopSize)+1;
   MutLoc=floor(rand*Nt)+1;  % Rand.# between 1 and Nt.
   NTPop(MutGene,MutLoc) = TMin(MutLoc) + TSpread(MutLoc)*rand;
end

% Non-uniform Parameter Mutation
for MutCt=1:round(PNonUniMut*Na*PPopSize);
   % For nullspace
   MutGene=floor(rand*PPopSize)+1;
   MutLoc=floor(rand*Na)+1;  % Rand.# between 1 and Na.
   OldValue=NAPop(MutGene,MutLoc);
   if rand<0.5
      Change = (AMax-OldValue)*(1-rand^((1-Pgen/PGens)^b));
    else
      Change = -(OldValue-AMin)*(1-rand^((1-Pgen/PGens)^b));
   end
   NAPop(MutGene,MutLoc) = OldValue + Change;
end

for MutCt=1:round(PNonUniMut*N*PPopSize);
   % For Qc
   MutGene=floor(rand*PPopSize)+1;
   MutLoc=floor(rand*Nc)+1;  % Rand.# between 1 and Nc.
   OldValue=NQcPop(MutGene,MutLoc);
   if rand<0.5
      Change = (QcMax(MutLoc)-OldValue)*(1-rand^((1-Pgen/PGens)^b));
    else
      Change = -(OldValue-QcMin(MutLoc))*(1-rand^((1-Pgen/PGens)^b));
   end
   NQcPop(MutGene,MutLoc) = OldValue + Change;

   % For T
```

```
        MutGene=floor(rand*PPopSize)+1;
        MutLoc=floor(rand*Nt)+1;  % Rand.# between 1 and Nt.
        OldValue=NTPop(MutGene,MutLoc);
        if rand<0.5
          Change = (TMax(MutLoc)-OldValue)*(1-rand^((1-Pgen/PGens)^b));
         else
          Change = -(OldValue-TMin(MutLoc))*(1-rand^((1-Pgen/PGens)^b));
        end
        NTPop(MutGene,MutLoc) = OldValue + Change;
      end

    end % IF Pgen < PGens
  end  % of Parameter GA

  % Now find the final answer
    bss=min(find(PFit==max(PFit)));
    Ybus = OYbus;
    Qc = NQcPop(bss,:)';
    Taps = NTPop(bss,:)';
    % Quantize Taps and Qc by rounding to the nearest 0.01
    Taps = round(100*Taps)/100;
    Qc = round(100*Qc)/100;
    QcList(QcInd) = Qc;

    if ~LF_Used(bss)
      v = NAPop(bss,:)'*NA2v;

      % Adjust Ybus & J to account for tap settings, if they've changed.
      % The turns ratio is 1:t and follows Debs (and is 1/c in Gross)
      if ~all(Taps==Taps0)
        dx = nJ0*v;    % To start, find x using the old nullspace.
        Delta = Delta0 + dx(1:N) - dx(SlackInd);
        Vm = Vm0 + dx(N+1:2*N);
        [Ybus,J2] = J_NewTaps(Delta,Vm,OYbus,J0,Taps0,Taps,TInd,TapY);
        Jr2 = J2(lss,:); nJ2 = null(Jr2);
        dx = nJ2*nJ2'*nJ0*v;
      else
        Ybus = OYbus;
        J2=J0;
        nJ2 = nJ0;
        dx = nJ0*v;
      end   % If taps have changed
      dx = dx + nJ2*nJ2(QcDV,:)'*(J2(QcInd+N,QcDV)\(Qc-Qc0));
      Delta = Delta0 + dx(1:N) - dx(SlackInd);
      Vm = Vm0 + dx(N+1:2*N);
    else % if ~LF_Used
      Delta = LF_D(bss,:)';
      Vm = LF_Vm(bss,:)';
      if ~all(Taps==Taps0)
        [Ybus,J2] = J_NewTaps(Delta,Vm,OYbus,J0,Taps0,Taps,TInd,TapY);
        Jr2 = J2(lss,:); nJ2 = null(Jr2);
      else
        Ybus = OYbus;
        J2=J0; nJ2=nJ0;
      end   % If taps have changed
    end % if ~LF_Used
```

```
  [Cost, NOx, Penalty, Pflow, Qflow, NQcList] = fitness(Delta,Vm,QcList,Ybus,Pd,Qd);
  Qc = NQcList(QcInd);
  Penalty = Pwt*Penalty;
  save answer
```

C.6 Fitness.m

```
function [Cost, NOx, Penalty, Pflow, Qflow, QcList] = fitness(Delta,Vm,OQcList,Ybus,Pd,Qd);
% Calculate the fitness of a particular solution.

global N Nc NumGenU CostCoeff NOxCoeff GenInd QcInd AMin AMax
global VmMin VmMax PMin PMax QMin QMax QcMin QcMax NOxMax Ptol LineMVA

QcList = OQcList;
[Fn_P, Fn_Q]=lf_eqs(Ybus, Delta, Vm, -1);
Pg = Fn_P + Pd;
Qg = Fn_Q + Qd - QcList;

dP=zeros(N,1); dQ=zeros(N,1); dQc=zeros(Nc,1);
Cost = 0;
NOx = 0;
for GenUnit=1:NumGenU
  UnitP = Pg(GenInd(GenUnit));
  UnitPMax = PMax(GenInd(GenUnit));
  Cost = Cost + polyval(CostCoeff(GenUnit,:),UnitP);
  NOx = NOx + UnitPMax*polyval(NOxCoeff(GenUnit,:),UnitP/UnitPMax);
end
pss1 = find(Pg < PMin); pss2 = find(Pg > PMax);
pss3 = find(Qg < QMin); pss4 = find(Qg > QMax);

dQ(pss3) = QMin(pss3)-Qg(pss3);
dQ(pss4) = -Qg(pss4)+QMax(pss4);
QcList(QcInd) = QcList(QcInd) + dQ(QcInd);
Qg = Fn_Q + Qd - QcList;
pss3 = find(Qg < QMin); pss4 = find(Qg > QMax);
dQ=zeros(N,1);

pss5 = find(QcList(QcInd) < QcMin); pss6 = find(QcList(QcInd) > QcMax);
dP(pss1) = PMin(pss1)-Pg(pss1);
dP(pss2) = -Pg(pss2)+PMax(pss2);
dQ(pss3) = QMin(pss3)-Qg(pss3);
dQ(pss4) = -Qg(pss4)+QMax(pss4);
dQc(pss5) = QcMin(pss5)-QcList(QcInd(pss5));
dQc(pss6) = -QcList(QcInd(pss6))+QcMax(pss6);
dP(find(abs(dP)<Ptol)) = 0;    % Ignore P errors smaller than Ptol.
dQ(find(abs(dQ)<Ptol)) = 0;
dQc(find(abs(dQc)<Ptol)) = 0;
Penalty = sum(abs(dP))+sum(abs(dQ))+sum(abs(dQc));
pss = find(Vm < VmMin); Penalty = Penalty + 10*sum(VmMin(pss)-Vm(pss));
pss = find(Vm > VmMax); Penalty = Penalty + 10*sum(Vm(pss)-VmMax(pss));
pss = find(Delta < AMin); Penalty = Penalty + sum(AMin-Delta(pss));
pss = find(Delta > AMax); Penalty = Penalty + sum(Delta(pss)-AMax);

% Penalize overloaded lines.  Note that the power-flow matrices are
```

```
%    close to symmetric.  Therefore, most overloads are penalized twice.
G=real(Ybus); B=imag(Ybus);
for B1=1:N
  for B2=1:N
    if B1~=B2
      T = G(B1,B2)*cos(Delta(B1)-Delta(B2)) + B(B1,B2)*sin(Delta(B1)-Delta(B2));
      U = G(B1,B2)*sin(Delta(B1)-Delta(B2)) - B(B1,B2)*cos(Delta(B1)-Delta(B2));
      Pflow(B1,B2) = -Vm(B1)*Vm(B1)*G(B1,B2) + Vm(B1)*Vm(B2)*T;
      Qflow(B1,B2) =  Vm(B1)*Vm(B1)*B(B1,B2) + Vm(B1)*Vm(B2)*U;
    else
      Pflow(B1,B1) = Vm(B1)*Vm(B1)*sum(G(B1,:));
      Qflow(B1,B1) = -Vm(B1)*Vm(B1)*sum(B(B1,:));
    end
  end
end
Sflow = sqrt(Pflow.^2 + Qflow.^2);
pss = find(Sflow > LineMVA);
Penalty = Penalty + 10*sum(sum(Sflow(pss)-LineMVA(pss)));

if NOx > NOxMax
  Penalty = Penalty + (NOx-NOxMax);
end
```

C.7 J_NewTaps.m

```
function [Ybus,J2] = J_NewTaps(Delta,Vm,OYbus,J0,OTaps,Taps,TInd,TapY)
% [Ybus2,J2] = J_NewTaps(Delta,Vm,OYbus,J0,OTaps,Taps,TInd,TapY)
% Finds the new Ybus and Jacobian when the taps are changed.

Ybus = OYbus;
N = size(Ybus,1);
dJ = zeros(size(J0));
dJs = zeros(4,4);

for ct=1:size(TInd,1);
  P = TInd(ct,1); S = TInd(ct,2);
  tr = [P S P+N S+N];        % Rows of J affected by taps
  Ot = OTaps(ct); t = Taps(ct);
  dYser = (Ot-t)*TapY(ct);
  dG = real(dYser); dB = imag(dYser);
  dYpp = (t*t-Ot*Ot)*TapY(ct);
  Ybus([P S],[P S]) = Ybus([P S],[P S]) + [dYpp dYser; dYser 0];

  Gs = dG*sin(Delta(P)-Delta(S));  Gc = dG*cos(Delta(P)-Delta(S));
  Bs = dB*sin(Delta(P)-Delta(S));  Bc = dB*cos(Delta(P)-Delta(S));
  Vp = Vm(P); Vs = Vm(S);

  % Now build the 4x4 submatrix that is affected by the taps.
  % dP/dd
  dJs(1,1) = Vp*Vs*(-Gs+Bc);   dJs(2,2) = Vp*Vs*( Gs+Bc);
  dJs(1,2) = Vp*Vs*( Gs-Bc);   dJs(2,1) = Vp*Vs*(-Gs-Bc);
  % dP/dV
  dJs(1,3) = Vs*(Gc+Bs) + 2*Vp*real(dYpp);  dJs(2,4) = Vp*(Gc-Bs);
  dJs(1,4) = Vp*(Gc+Bs);                    dJs(2,3) = Vs*(Gc-Bs);
  % dQ/dd
  dJs(3,1) = Vs*Vp*( Gc+Bs);   dJs(4,2) = Vs*Vp*( Gc-Bs);
```

```
  dJs(3,2) = Vs*Vp*(-Gc-Bs);   dJs(4,1) = Vs*Vp*(-Gc+Bs);
  % dQ/dV
  dJs(3,3) = Vs*(Gs-Bc) - 2*Vp*imag(dYpp);  dJs(4,4) = Vp*(-Gs-Bc);
  dJs(3,4) = Vp*(Gs-Bc);                    dJs(4,3) = Vs*(-Gs-Bc);
  % Use the 4x4 submatrix to update the Jacobian
  dJ(tr,tr) = dJ(tr,tr) + dJs;
end
J2 = J0+dJ;
```

C.8 LF_Eqs.m

```
function [Fn_P, Fn_Q]=lf_eqs(Ybus, Delta, Vm, NumPQspec)
% [Fn_P, Fn_Q] = LF_Eqs(Ybus, Delta, Vm, NumPQ)
%    The Load Flow Equations calculate the real power at all PQ and PV buses,
%    and the reactive power at all PQ buses,
%    given the magnitude & angle of all bus voltages.
%
% Ybus is the bus admittance matrix.
% Delta and Vm are the vectors of angle and voltage magnitude for all buses.
%    The elements of Ybus, Delta, and Vm must be ordered so that Bus #1 is the Slack bus,
%    Buses 2 through (NumPQ+1) are the PQ buses, and the remaining buses are PV buses.
% NumPQ is the number of PQ buses.
%
% Fn_P is a vector of the real power injected into the PQ and PV buses
% Fn_Q is a vector of the reactive power injected into the PQ buses
% The rows of Fn_P and Fn_Q are in the same order as in Delta and Vm.
%
% If NumPQ is set to -1, the function calculates P and Q at ALL buses.

G=real(Ybus); B=imag(Ybus);
N = length(Vm);      % Total number of buses

% If NumPQ is set to -1, we need to find P and Q for buses 1 through N.
%    We set NumPQ to N in this case, to find Q at all nodes.
%    We subtract 1 from the Bus index, to allow us to start at Bus 1, instead of Bus 2.
% Otherwise, we need P for Buses 2 through N, and Q for buses 2 through (NumPQ+1).
%    In this case, All is set to 0, and it can be ingored.
if NumPQspec==-1
  All=1; NumPQ=N;    % "All" is the amount to adjust the Bus indices
 else
  All=0; NumPQ=NumPQspec;
end

% Initialize the output vectors
Fn_P = zeros(N-1+All, 1);
Fn_Q = zeros(NumPQ, 1);

% Form Fn_P
for r=1:N-1+All       % The current row of Fn_P
  Br = r+1-All;      % The corresponding bus number
  Sum = Vm(Br)*Vm(Br)*G(Br,Br);
  for m=1:N
    if m~=Br
      T = G(Br,m)*cos(Delta(Br)-Delta(m)) + B(Br,m)*sin(Delta(Br)-Delta(m));
      Sum = Sum + Vm(Br)*Vm(m)*T;
    end
```

```
  end
  Fn_P(r)=Sum;
end

% Form Fn_Q
for r=1:NumPQ          % The current row of Fn_Q
  Br = r+1-All;       % The corresponding bus number
  Sum = -Vm(Br)*Vm(Br)*B(Br,Br);
  for m=1:N
    if m~=Br
      U = G(Br,m)*sin(Delta(Br)-Delta(m)) - B(Br,m)*cos(Delta(Br)-Delta(m));
      Sum = Sum + Vm(Br)*Vm(m)*U;
    end
  end
  Fn_Q(r)=Sum;
end
```

C.9 LF_Jacob.m

```
function J=lf_jacob(Ybus, Delta, Vm, NumPQ)
% J = LF_Jacob(Ybus, Delta, Vm, NumPQ)
%    calculates the Load-flow Jacobian matrix, for use with the Newton-Raphson method.
%
% Ybus is the bus admittance matrix.
% Delta and Vm are the vectors of angle and voltage magnitude for all buses.
%    The elements of Ybus, Delta, and Vm must be ordered so that Bus #1 is the Slack bus,
%    Buses 2 through (NumPQ+1) are the PQ buses, and the remaining buses are PV buses.
% NumPQ is the number of PQ buses.
%
% J is the Jacobian.  Its rows are in the same order as in Vm and Delta.

G=real(Ybus); B=imag(Ybus);
N = length(Vm);      % Total number of buses

% Initialize J11, J12, J21, and J22, the submatrices of the Jacobian.
J11 = zeros(N-1, N-1);
J12 = zeros(N-1, NumPQ);
J21 = zeros(NumPQ, N-1);
J22 = zeros(NumPQ, NumPQ);

% Form J11
for r=1:N-1      % Row of the submatrix
  for c=1:N-1   % Column of the submatrix
    Br = r+1; Bc = c+1;    % Buses corresponding to this row & column of J11
    if r==c
      Sum=0;
      for m=1:N
        if m~=Br
          U = G(Br,m)*sin(Delta(Br)-Delta(m)) - B(Br,m)*cos(Delta(Br)-Delta(m));
          Sum = Sum - Vm(Br)*Vm(m)*U;
        end
      end
      J11(r,c)=Sum;
    else
      U = G(Br,Bc)*sin(Delta(Br)-Delta(Bc)) - B(Br,Bc)*cos(Delta(Br)-Delta(Bc));
```

```
      J11(r,c) = Vm(Br)*Vm(Bc)*U;
    end
  end
end

% Form J12
for r=1:N-1          % Row of the submatrix
  for c=1:NumPQ      % Column of the submatrix
    Br = r+1;  Bc = c+1;   % Buses corresponding to this row & column of J12
    if r==c
      Sum = 2*Vm(Br)*G(Br,Br);
      for m=1:N
        if m~=Br
          T = G(Br,m)*cos(Delta(Br)-Delta(m)) + B(Br,m)*sin(Delta(Br)-Delta(m));
          Sum = Sum + Vm(m)*T;
        end
      end
      J12(r,c)=Sum;
    else
      T = G(Br,Bc)*cos(Delta(Br)-Delta(Bc)) + B(Br,Bc)*sin(Delta(Br)-Delta(Bc));
      J12(r,c) = Vm(Br)*T;
    end
  end
end

% Form J21
for r=1:NumPQ      % Row of the submatrix
  for c=1:N-1      % Column of the submatrix
    Br = r+1;  Bc = c+1;   % Buses corresponding to this row & column of J21
    if r==c
      Sum = 0;
      for m=1:N
        if m~=Br
          T = G(Br,m)*cos(Delta(Br)-Delta(m)) + B(Br,m)*sin(Delta(Br)-Delta(m));
          Sum = Sum + Vm(Br)*Vm(m)*T;
        end
      end
      J21(r,c)=Sum;
    else
      T = G(Br,Bc)*cos(Delta(Br)-Delta(Bc)) + B(Br,Bc)*sin(Delta(Br)-Delta(Bc));
      J21(r,c) = -Vm(Br)*Vm(Bc)*T;
    end
  end
end

% Form J22
for r=1:NumPQ      % Row of the submatrix
  for c=1:NumPQ    % Column of the submatrix
    Br = r+1;  Bc = c+1;   % Buses corresponding to this row & column of J22
    if r==c
      Sum = -2*Vm(Br)*B(Br,Br);
      for m=1:N
        if m~=Br
          U = G(Br,m)*sin(Delta(Br)-Delta(m)) - B(Br,m)*cos(Delta(Br)-Delta(m));
          Sum = Sum + Vm(m)*U;
        end
```

```
        end
        J22(r,c)=Sum;
      else
        U = G(Br,Bc)*sin(Delta(Br)-Delta(Bc)) - B(Br,Bc)*cos(Delta(Br)-Delta(Bc));
        J22(r,c) = Vm(Br)*U;
      end
    end
end

J = [J11 J12; J21 J22];
```

C.10 OPF.m

```
function [Vm,Delta, SlackP, SlackQ, Pflow,Qflow] =
OPF(Ybus,NodeList,BusTypes,Pdata,Qdata,Vdata,SlackAng,GuessDelta,GuessVm)
% [Vm,Delta, SlackP, SlackQ, Pflow,Qflow] =
OPF(Ybus,NodeList,BusTypes,Pdata,Qdata,Vdata,SlackAng,GuessDelta,GuessVm)
%
% Pg and Qg have one element for each generator.
% Newton-Raphson solution of OPF.
% GuessDelta and GuessVm are optional initial guesses.

% Find the slack bus
SlackInd = find(BusTypes==1);  % Index of the slack bus
VmSlack = Vdata(SlackInd);
DeltaSlack = SlackAng;

% Find the indices to the PQ buses, and their given P and Q.
PQbus  = find(BusTypes==2);   % The indices of the PQ buses
PQbusP = Pdata(PQbus);
PQbusQ = Qdata(PQbus);

% Find the indices to the PV buses, and their given P and V.
PVbus  = find(BusTypes==3);   % The indices of the PV buses
PVbusP = Pdata(PVbus);
PVbusV = Vdata(PVbus);

N=length(NodeList);
NumPQ=length(PQbus);
NumPV=N-NumPQ-1;

% Now sort the buses so that Bus #1 is the Slack bus, Buses 2 through "NumPQ+1"
%   are the PQ buses, and the remaining buses are the PV buses.
SortOrder = [SlackInd PQbus PVbus];
OldYbus=Ybus;
Ybus = Ybus(SortOrder,SortOrder);   % Rearrange Ybus to account for the new ordering.
NodeSort = NodeList(SortOrder);

% Identify the given quantities
Pg = [PQbusP; PVbusP];
Qg = PQbusQ;
Vg = PVbusV;
Vm = zeros(N,1); Delta=zeros(N,1);
Vm(1)=VmSlack; Delta(1)=DeltaSlack;  % Slack bus Voltage magnitude and angle.
Vm((NumPQ+2):N)=Vg;    % Voltage magnitudes at the PV buses.
```

```
% Now start the Newton-Raphson algorithm
Tol=1e-5; % Tolerance for the solution
Iter=0;
IterLimit=20;   % Limit the number of iterations, to prevent an infinite loop.

% If no guess is specified, use a flat start for the initial guess of the unknowns
if nargin==7
  GuessDelta=zeros(N-1, 1);
  GuessVm = ones(NumPQ,1);
end

Done=0;
while ~Done
  if Iter==IterLimit
    Done=1;
  end

  Delta(2:N)=GuessDelta;
  Vm(2:NumPQ+1)=GuessVm;
  [Fn_P, Fn_Q] = LF_Eqs(Ybus, Delta, Vm, NumPQ);
  % Calculate the mismatch in P & Q
  ErrP = Pg - Fn_P;
  ErrQ = Qg - Fn_Q;
  Err = [ErrP; ErrQ];

  if max(abs([ErrP; ErrQ])) <= Tol   % Have we converged?
    Done=1;
   else        % If not converged, iterate again.
    J = LF_Jacob(Ybus, Delta, Vm, NumPQ);
    Jinv = inv(J);
    if rcond(J)<1e-8
      if Iter==0
        GuessDelta=zeros(N-1,1);
        GuessVm = ones(NumPQ,1);
        Err = zeros(N+NumPQ-1,1);
      else
        Done=1;
      end
    end
    Iter = Iter+1;
    GuessDelta = GuessDelta + Jinv(1:N-1,:)*Err;
    GuessVm = GuessVm + Jinv(N:end,:)*Err;
  end
end

% Delta and Vm are now the solutions to the LF equations.
% Use the load-flow equations one last time to find P & Q at ALL buses.
[P, Q] = LF_Eqs(Ybus, Delta, Vm, -1);

% Now find the real & reactive power flowing from each bus toward each other bus.
% The results will be stored in matrices Pflow and Qflow.
% Let B1 and B2 be two buses.  If B1 is not equal to B2:
%    Pflow(B1,B2) = real power flowing from B1 toward B2.
%    Qflow(B1,B2) = reactive power flowing from B1 toward B2.
% The main diagonal entries of these matrices are defined as the shunt power.
```

```
%    For example, Pflow(B1,B1) = real power flowing from bus B1 to the ground.
G=real(Ybus); B=imag(Ybus);
for B1=1:N
  for B2=1:N
    if B1~=B2
      T = G(B1,B2)*cos(Delta(B1)-Delta(B2)) + B(B1,B2)*sin(Delta(B1)-Delta(B2));
      U = G(B1,B2)*sin(Delta(B1)-Delta(B2)) - B(B1,B2)*cos(Delta(B1)-Delta(B2));
      Pflow(B1,B2) = -Vm(B1)*Vm(B1)*G(B1,B2) + Vm(B1)*Vm(B2)*T;
      Qflow(B1,B2) =  Vm(B1)*Vm(B1)*B(B1,B2) + Vm(B1)*Vm(B2)*U;
    else
      Pflow(B1,B1) = Vm(B1)*Vm(B1)*sum(G(B1,:));
      Qflow(B1,B1) = -Vm(B1)*Vm(B1)*sum(B(B1,:));
    end
  end
end

% The buses are sorted to be in numerical order.
[SortedBuses, NumOrder]=sort(NodeSort);

SlackP = P(1); % P and Q have slack as bus #1.
SlackQ = Q(1);
Vm=Vm(NumOrder);
Delta=Delta(NumOrder);
Pflow=Pflow(NumOrder,NumOrder);
Qflow=Qflow(NumOrder,NumOrder);
```

C.11 OPF_Jacb.m

```
function J=opf_jacb(Ybus, Delta, Vm)
% J = OPF_Jacb(Ybus, Delta, Vm)
%    calculates the Load-flow Jacobian matrix, for use with the Newton-Raphson method.
%
% Ybus is the bus admittance matrix.
% Delta and Vm are the vectors of angle and voltage magnitude for all buses.
% J is the Jacobian.  Its rows are in the same order as in Vm and Delta.

G=real(Ybus); B=imag(Ybus);
N = length(Vm);      % Total number of buses

% Initialize J11, J12, J21, and J22, the submatrices of the Jacobian.
J11 = zeros(N, N);
J12 = zeros(N, N);
J21 = zeros(N, N);
J22 = zeros(N, N);

% Form J11
for r=1:N      % Row of the submatrix
  for c=1:N   % Column of the submatrix
    Br = r; Bc = c;    % Buses corresponding to this row & column of J11
    if r==c
      Sum=0;
      for m=1:N
        if m~=Br
          U = G(Br,m)*sin(Delta(Br)-Delta(m)) - B(Br,m)*cos(Delta(Br)-Delta(m));
          Sum = Sum - Vm(Br)*Vm(m)*U;
        end
```

```
        end
        J11(r,c)=Sum;
      else
        U = G(Br,Bc)*sin(Delta(Br)-Delta(Bc)) - B(Br,Bc)*cos(Delta(Br)-Delta(Bc));
        J11(r,c) = Vm(Br)*Vm(Bc)*U;
      end
    end
end

% Form J12
for r=1:N          % Row of the submatrix
  for c=1:N     % Column of the submatrix
    Br = r;  Bc = c;   % Buses corresponding to this row & column of J12
    if r==c
      Sum = 2*Vm(Br)*G(Br,Br);
      for m=1:N
        if m~=Br
          T = G(Br,m)*cos(Delta(Br)-Delta(m)) + B(Br,m)*sin(Delta(Br)-Delta(m));
          Sum = Sum + Vm(m)*T;
        end
      end
      J12(r,c)=Sum;
    else
      T = G(Br,Bc)*cos(Delta(Br)-Delta(Bc)) + B(Br,Bc)*sin(Delta(Br)-Delta(Bc));
      J12(r,c) = Vm(Br)*T;
    end
  end
end

% Form J21
for r=1:N      % Row of the submatrix
  for c=1:N     % Column of the submatrix
    Br=r;  Bc=c;   % Buses corresponding to this row & column of J21
    if r==c
      Sum = 0;
      for m=1:N
        if m~=Br
          T = G(Br,m)*cos(Delta(Br)-Delta(m)) + B(Br,m)*sin(Delta(Br)-Delta(m));
          Sum = Sum + Vm(Br)*Vm(m)*T;
        end
      end
      J21(r,c)=Sum;
    else
      T = G(Br,Bc)*cos(Delta(Br)-Delta(Bc)) + B(Br,Bc)*sin(Delta(Br)-Delta(Bc));
      J21(r,c) = -Vm(Br)*Vm(Bc)*T;
    end
  end
end

% Form J22
for r=1:N      % Row of the submatrix
  for c=1:N   % Column of the submatrix
    Br = r;  Bc = c;   % Buses corresponding to this row & column of J22
    if r==c
      Sum = -2*Vm(Br)*B(Br,Br);
      for m=1:N
```

```
          if m~=Br
            U = G(Br,m)*sin(Delta(Br)-Delta(m)) - B(Br,m)*cos(Delta(Br)-Delta(m));
            Sum = Sum + Vm(m)*U;
          end
        end
        J22(r,c)=Sum;
      else
        U = G(Br,Bc)*sin(Delta(Br)-Delta(Bc)) - B(Br,Bc)*cos(Delta(Br)-Delta(Bc));
        J22(r,c) = Vm(Br)*U;
      end
    end
end

J = [J11 J12; J21 J22];
```

C.12 FDLF.m

```
function [Vm,Delta, SlackP, SlackQ, Pflow,Qflow] =
FDLF(Ybus,NodeList,BusTypes,Pdata,Qdata,Vdata,SlackAng,GuessDelta,GuessVm)
%
% Pg and Qg have one element for each generator.
% Fast-decoupled load-flow solution of OPF.
% GuessDelta and GuessVm are optional initial guesses.

% Find the slack bus
SlackInd = find(BusTypes==1);  % Index of the slack bus
VmSlack = Vdata(SlackInd);
DeltaSlack = SlackAng;

% Find the indices to the PQ buses, and their given P and Q.
PQbusList  = find(BusTypes==2);   % The indices of the PQ buses
PQbusP = Pdata(PQbusList);
PQbusQ = Qdata(PQbusList);

% Find the indices to the PV buses, and their given P and V.
PVbusList  = find(BusTypes==3);   % The indices of the PV buses
PVbusP = Pdata(PVbusList);
PVbusV = Vdata(PVbusList);

N=length(NodeList);
NumPQ=length(PQbusList);
NumPV=N-NumPQ-1;

% Find the positions of the PQ buses in NodeList
PQbus=zeros(NumPQ,1);
for c=1:NumPQ
  NextPQ = find(NodeList==PQbusList(c));
  if length(NextPQ)~=1
    error('Each element of PQbusList must appear in NodeList exactly once.')
  end
  PQbus(c) = NextPQ;
end

% Find the positions of the PV buses in NodeList
PVbus=zeros(NumPV,1);
for c=1:NumPV
```

```
  NextPV = find(NodeList==PVbusList(c));
  if length(NextPV)~=1
    error('Each element of PVbusList must appear in NodeList exactly once.')
  end
  PVbus(c) = NextPV;
end

% Now sort the buses so that Bus #1 is the Slack bus, Buses 2 through "NumPQ+1" are
%   the PQ buses, and the remaining buses are the PV buses.
% Ybus is rearranged to account for the new ordering.
SortOrder = [SlackInd PQbusList PVbusList];
%NodeSort = NodeList(SortOrder);

% Identify the given quantities
Pg = [PQbusP; PVbusP];
Qg = PQbusQ;
Vg = PVbusV;
Vm = zeros(N,1); Delta=zeros(N,1);

Vm(SortOrder(1))=VmSlack; Delta(SortOrder(1))=DeltaSlack;  % Slack bus Voltage magnitude
and angle.
Vm(SortOrder((NumPQ+2):N))=Vg;    % Voltage magnitudes at the PV buses.

% Now start the Fast Decoupled Load Flow algorithm
Tol=1e-3; % Tolerance for the solution
Iter=0;
IterLimit=15;   % Limit the number of iterations, to prevent an infinite loop.

% Form the FDLF matrices B' and B".
% Note that we delete the first row & column of B', since Delta is known at
%    the slack bus.
% Similarly, we only keep the submatrix of B" corresponding to PQ buses.
B = imag(Ybus);
Bprime = -B(SortOrder(2:N),SortOrder(2:N));   % B' = -B, except on the main diagonal.
for ct=2:N              % Now correct the main diagonal of B'
  Bprime(ct-1,ct-1) = sum(B(SortOrder(ct),:)) - B(SortOrder(ct),SortOrder(ct));
    % Since the sum is supposed to exculde B(ct,ct), I subtract it back out
    %   of the full sum.
end
B2prime = -B(SortOrder(2:NumPQ+1),SortOrder(2:NumPQ+1));  % B" = -B, except on the main
diagonal.
for ct=2:NumPQ+1                % Now correct the diagonal of B"
  B2prime(ct-1,ct-1) = -sum(B(SortOrder(ct),:)) - B(SortOrder(ct),SortOrder(ct));
end
% Now invert B' and B"
InvBp = inv(Bprime);
InvB2p = inv(B2prime);

% If no guess is specified, use a flat start for the initial guess of the unknowns
if nargin < 8
  GuessDelta=zeros(N-1, 1);
  GuessVm = ones(NumPQ,1);
end

Delta(SortOrder(2:N))=GuessDelta;   % Delta & Vm are the angle & magnitude at ALL buses,
Vm(SortOrder(2:NumPQ+1))=GuessVm;   % not just where these quantities are unknown.
```

```
Done=0;
while ~Done
  if Iter==IterLimit
    %error('Iteration limit reached before convergence.')
    Done = 1;
  end

  [Fn_P, Fn_Q] = LF_Eqs(Ybus, Delta, Vm, -1);
  % Calculate the mismatch in P & Q
  ErrP = Pg - Fn_P([PQbusList PVbusList]);
  ErrQ = Qg - Fn_Q(PQbusList);
  Err = [ErrP; ErrQ];

  if max(abs([ErrP; ErrQ])) <= Tol   % Have we converged?
    Done=1;
   else        % If not converged, iterate again.
    Iter = Iter+1;

    % Update the angles as soon as the new values are available.
    GuessDelta = GuessDelta + InvBp*(ErrP./Vm(SortOrder(2:N)));
    Delta(SortOrder(2:N))=unwrap(GuessDelta);

    % Now recompute the reactive power mismatch.
    % (LF_Eqs also returns the computed real power, which is ignored here.)
    [Fn_P, Fn_Q] = LF_Eqs(Ybus, Delta, Vm, -1);
    ErrQ = Qg - Fn_Q(PQbusList);

    % Now update the Voltage magnitudes at the PQ buses.
    GuessVm = GuessVm + InvB2p*(ErrQ./Vm(SortOrder(2:NumPQ+1)));
    Vm(SortOrder(2:NumPQ+1))=GuessVm;
  end
end

% Delta and Vm are now the solutions to the LF equations.
% Use the load-flow equations one last time to find P & Q at ALL buses.
[P, Q] = LF_Eqs(Ybus, Delta, Vm, -1);

% Now find the real & reactive power flowing from each bus toward each other bus.
% The results will be stored in matrices Pflow and Qflow.
% Let B1 and B2 be two buses.  If B1 is not equal to B2:
%    Pflow(B1,B2) = real power flowing from B1 toward B2.
%    Qflow(B1,B2) = reactive power flowing from B1 toward B2.
% The main diagonal entries of these matrices are defined as the shunt power.
%    For example, Pflow(B1,B1) = real power flowing from bus B1 to the ground.
G=real(Ybus); B=imag(Ybus);
for ctB1=1:N
  for ctB2=1:N
    B1=SortOrder(ctB1);
    B2=SortOrder(ctB2);
    if B1~=B2
      T = G(B1,B2)*cos(Delta(B1)-Delta(B2)) + B(B1,B2)*sin(Delta(B1)-Delta(B2));
      U = G(B1,B2)*sin(Delta(B1)-Delta(B2)) - B(B1,B2)*cos(Delta(B1)-Delta(B2));
      Pflow(B1,B2) = -Vm(B1)*Vm(B1)*G(B1,B2) + Vm(B1)*Vm(B2)*T;
      Qflow(B1,B2) =  Vm(B1)*Vm(B1)*B(B1,B2) + Vm(B1)*Vm(B2)*U;
     else
```

```
        Pflow(B1,B1) = Vm(B1)*Vm(B1)*sum(G(B1,:));
        Qflow(B1,B1) = -Vm(B1)*Vm(B1)*sum(B(B1,:));
      end
    end
end

SlackP = P(SortOrder(1)); % P and Q have slack as bus #1.
SlackQ = Q(SortOrder(1));
```

Vita

Reid S. Maust

Reid Maust earned his B.S. degree in Electrical Engineering from West Virginia University in 1992. He served as a graduate research assistant and earned his M.S. degree in Electrical Engineering from West Virginia University in 1993, specializing in communications. During his Ph.D. studies at West Virginia University, Reid has served as graduate teaching assistant and graduate research assistant, specializing in automatic control. He intends to adapt the GA-OPF algorithm presented here for use by an electric utility.