To the Graduate Council:

I am submitting herewith a thesis written by Jovan Ilić entitled "Artificial Intelligence Methods in Deregulated Power Systems Operations". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Electrical Engineering.

Jack Lawler, Major Professor

We have read this thesis
and recommend its acceptance:

Marshall O. Pace

Robert E. Uhrig

Leon M. Tolbert

Accepted for the Council:

Interim Vice Provost and
Dean of The Graduate School

Artificial Intelligence Methods in Deregulated Power Systems Operations

A Dissertation

Presented for the

Doctor of Philosophy

The University of Tennessee, Knoxville

Jovan Ilić

May 2001

Acknowledgments

My thanks go to Dr. Jack Lawler for his help, support, encouragement, infinite patience and availability at the most critical times. Also, I would like to thank my committee members: Drs. Marshall Pace, Robert Uhrig, Leon Tolbert and Hairong Qi for their most helpful feedback and constructive questioning of this research. My gratitude also goes to Dr. Marija Ilić for her invaluable suggestions and the endless research she has done in the power systems area. Last, but not least, thanks go to my wife, Winkie, for her unconditional love and understanding for my many sleepless nights and also to my dear friend, Mr. Tom Rohtert, for supporting what at times looked an endless endeavor.

During this research, the work of the author was supported in part by the Electric Power Research Institute and the US Army Research Office. The EPRI managers on this project were Drs. Masoud Amin and Ram Adapa. This support is gratefully acknowledged. In addition, the author would like to thank Dale Bradshaw, Manager of Power Delivery Technologies for TVA Energy Research and Technology Applications. Mr. Bradshaw provided many helpful suggestions in shaping the direction of this work.

Abstract

With the introduction of the power systems deregulation, many classical power transmission and distribution optimization tools became inadequate. Optimal Power Flow and Unit Commitment are common computer programs used in the regulated power industry. This work is addressing the Optimal Power Flow and Unit Commitment in the new deregulated environment. Optimal Power Flow is a high dimensional, non–linear, and non–convex optimization problem. As such, it is even now, after forty years since its introduction, a research topic without a widely accepted solution able to encompass all areas of interest. Unit Commitment is a high dimensional, combinatorial problem which should ideally include the Optimal Power Flow in its solution. The dimensionality of a typical Unit Commitment problem is so great that even the enumeration of all the combinations would take too much time for any practical purposes.

This dissertation attacks the Optimal Power Flow problem using non–traditional tools from the Artificial Intelligence arena. Artificial Intelligence optimization methods are based on stochastic principles. Usually, stochastic optimization methods are successful where all other classical approaches fail. We will use Genetic Programming optimization for both Optimal Power Flow and Unit Commitment. Long processing times will also be addressed through supervised machine learning.

Contents

1 Introduction **1**
- 1.1 Independent System Operators . . . 3
- 1.2 Software and Hardware Tools Required for Free Market Implementation . . . 9
- 1.3 Dissertation Objective . . . 13

2 Genetic Algorithm Optimization **18**
- 2.1 Generic Genetic Algorithms . . . 18
- 2.2 Genetic Algorithms Applied to Power Systems . . . 21

3 Genetic Algorithm Based Optimal Power Flow and Unit Commitment **29**
- 3.1 Power Flow Implementation . . . 30
- 3.2 Genetic Algorithm Based Optimal Power Flow . . . 34
- 3.3 GAOPF Operational Principles . . . 37
- 3.4 GAOPF Examples . . . 38
  - 3.4.1 Power Cost Minimizing OPF . . . 40
  - 3.4.2 Power Losses Minimization . . . 42
  - 3.4.3 Minimizing the Production Cost and Power Losses . . . 43
  - 3.4.4 Minimizing Production Cost and Controlling Line Congestion . . . 46
- 3.5 Optimal Power Flow Summary . . . 50
- 3.6 Genetic Algorithm Based Unit Commitment . . . 55
- 3.7 Genetic Algorithm Drawbacks and Workarounds . . . 56

4 Decision Tree Theory **61**
- 4.1 Decision Tree Inductive Learning . . . 61
- 4.2 Decision Tree Example . . . 63
- 4.3 Decision Tree Issues . . . 67

5 GAOPF Based Unit Commitment and Machine Learning **71**
- 5.1 Genetic Algorithm Unit Commitment . . . 73
- 5.2 Machine Learning of the Unit Commitment . . . 76
- 5.3 Analysis of Genetic Algorithm Unit Commitment . . . 80

6 Conclusions and Future Research **82**
6.1 Conclusions 84
6.2 Future Work 87

Bibliography **89**

Appendices **93**

A Power Flow Review **94**

B Optimal Power Flow Review **100**
B.1 Sequential Quadratic Programming Optimal Power Flow 102

C Unit Commitment **105**
C.1 Classical Methods 106
C.1.1 Priority-List Method 108
C.1.2 Dynamic Programming Solution 109
C.1.3 The Lagrange Relaxation Solution 111
C.1.4 Unit Commitment in the Deregulated Market 114

D Programs Code **116**
D.1 Genetic Algorithm OPF 117
D.2 GAOPF Based Unit Commitment 156
D.3 Decision Tree Machine Learning 170

Vita **199**

List of Tables

3.1 GAOPF Optimization of the production cost. 40
3.2 GAOPF Optimization of the production cost. 44
3.3 SQP refinement of the GAOPF power cost optimization. 45
3.4 GAOPF power losses optimization. 47
3.5 SQP refinement of GAOPF power losses optimization. 49
3.6 GAOPF power cost and losses optimization. 51
3.7 SQP refinement of GAOPF power cost and losses optimization. 52
3.8 GAOPF power cost minimization and line congestion control. 54
3.9 OPF Summary with Power Cost Minimization. 54
3.10 OPF Summary with Power Losses Minimization. 55
3.11 OPF Summary with Power Cost and Power Losses Minimization. 55
3.12 Units available for scheduling. 56
3.13 Simple unit commitment over a 24hr period. 57
3.14 Unit commitment with minimum up–time over a 24hr period. 58

5.1 Network buses. 74
5.2 Network generators. 75
5.3 Unit Commitment as Seen by an ISO. 76
5.4 Unit Commitment Schedule With Critical Loads. 78

C.1 Number of possible Unit Commitment combinations over 24 hours period. 107

List of Figures

1.1 U.S. power system interconnection. 4

2.1 Genetic algorithm steps. 22
2.2 Voltage penalty function example. 26

3.1 Power flow data structure diagram. 31
3.2 Power World 7.0 output. 39
3.3 Power World 5.0 output. 41
3.4 Genetic Algorithm Convergence Rate. 42
3.5 Power World 7.0 minimizing production cost while enforcing line flow limits. 48
3.6 Daily load as a function of time. 59
3.7 Distributed GAOPF space time dependence. 60

4.1 Decision tree induced from the training data. 66
4.2 Decision Tree Classification Surface Approximation. 69

5.1 Genetic algorithm OPF based unit commitment. 72

Chapter 1

Introduction

Currently, in the U.S., there are 3,195 electric utilities capable of providing more than 776,000 megawatts of power [3]. Of these 3,195 utilities, two thirds have no generating capability. The wholesale power market can affect both the generation and retail markets. An analysis done by the Energy Information Administration (EIA) indicates that the market is and will continue to pay high premiums for assurance of supply [10]. The same study showed that industrial customers paid nearly wholesale price during the period from 1992 through 1996, with a small decline in price. At the same time, the retail customers' price increased slightly. The U.S. government has recognized the benefits of a free power market and the possibilities of competitive power production and transmission. There is a number of policies introduced since the mid 1970's to allow non utility generators to participate in power production.

In 1978, the Public Utility Regulatory Policies Act (PURPA) was designed to en-

courage the supply of wholesale power to electric utilities from renewable energy sources. PURPA regulations preserved the industry's vertical structure, but it was accepted by the utilities with mixed feelings.

The Energy Policy Act of 1992 (EPACT) was introduced to accelerate competition at the wholesale level. EPACT also amended the Public Utility Holding Company Act (PUHCA) of 1935, the Federal Power Act of 1935 and PURPA provisions related to the future of non–utility power generations and the associated power transactions. The two main changes made by EPACT to improve competition were to create a new class of exempt wholesale generators (EWGs) and to expand the authority of the Federal Energy Regulatory Commission (FERC) to order open transmission access under Section 211 of the Federal Power Act. FERC was entrusted to determine the EWG status and to ensure the availability of transmission facilities on a non–discriminatory basis.

FERC's most important rulings are Order No. 888 and Order No. 889, issued in April of 1996. These orders require all public utilities that own, control or operate transmission facilities to file non-discriminatory open-access tariffs that would be applied to all parties contracting for transmission service. After these two orders were issued, open market forces increased on both state and federal levels.

The details of deregulation are left to the states, and therefore there is no uniform approach, and the pace of deregulation varies across the country. It is reasonable to address the most common problems before the specific ones. First, we will review the most common scenario, then identify the problem of interest, review existing approaches,

and propose a better and more flexible solution.

1.1 Independent System Operators

The electric power industry is composed of power generation, power transmission grids and distribution power grids. Power generation represents 55% of major investor owned utilities' plant investment, transmission represents 12%, and the distribution 29% according to [3]. For the competitive market to function freely, the transmission grid is of the highest importance.

The U.S. power transmission system is serviced by three of the five power grids in North America: the Eastern Interconnected System, the Western Interconnected System and the Texas Interconnected System as shown in Figure 1.1 [3]. The other two networks have limited interconnections to the U.S. All U.S. utilities are interconnected with at least one other utility by these major grids, except Alaska and Hawaii. The interconnections are divided into 152 regional control areas that monitor and control a regional transmission grid. Control areas are responsible for the reliable operation of the transmission system, unit commitment and economic dispatch. To improve efficiency, some utilities have created regional power pools to coordinate the generation and transmission among their members. There are 22 centrally dispatched power pools in the United States.

Power generation has become more competitive during the last twenty years because of the technological advances and federal and state encouragement of free competition.

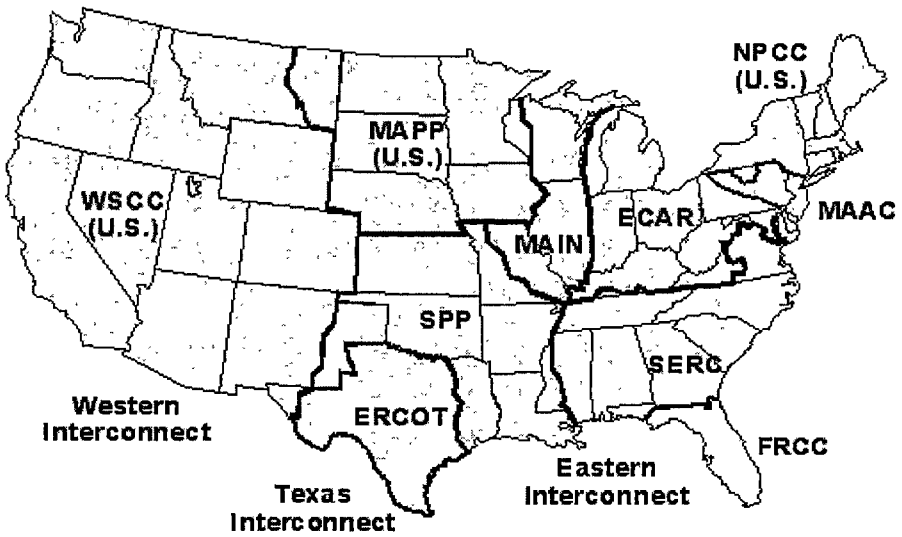


Figure 1.1: U.S. power system interconnection.

However, only after FERC's Order 888 were the transmission owners required to provide the transmission capability to other generators at the same terms they provide to themselves. To enforce this principle, a number of regulators and industry members insisted on separation of transmission ownership and transmission control.

In 1994, two California utilities (San Diego Gas and Electric Company and Southern California Edison Company) proposed a regional company that would have operating control of some or all generators and all transmission facilities. This concept developed into what is known as the Independent System Operator (ISO). Since 1994, supported by FERC's orders 888 and 889, ISOs are being introduced around the United States.

The ISO idea is intended to eliminate discriminatory practices and reduce self-dealing and other power market abuses. The ISO should also be able to develop efficient

methods for pricing of transmission services, resulting in lower transmission costs to customers. Because of its operational oversight of a large transmission system, the ISO is also in a position to resolve congestion more efficiently than the utilities or regional power pools. ISOs can also provide one-stop shopping for transmission services as well as timely and objective resolutions of disputes among utilities.

The ISO concept is still in the development stage and is not without criticism. There are still industry members which believe that vertically integrated utilities can manipulate the market and that the transmission and generation should be separate entities. Regardless of whether or not this kind of separation will happen, ISOs' success depends on the fact that it is a non-profit entity that controls but does not own the transmission facilities. Another issue is whether ISOs will have sufficient control over the transmission system to be able to provide non-discriminatory services. FERC Order 888 provides the following guidelines for ISOs [3]:

- The ISO's governance should be structured in a fair and non–discriminatory manner.
- An ISO and its employees should have no financial interest in the economic performance of any power market participant. An ISO should adopt and enforce strict conflict–of–interest standards.
- An ISO should provide open access to the transmission system and all services under its control at non–pancaked rates pursuant to a single, unbundled, grid–wide tariff that applies to all eligible users.
- An ISO should have the primary responsibility in ensuring short–term reliability of grid operations. Its role should be well defined and comply with applicable standards set by the North American Reliability Council and the regional reliability council.
- An ISO should have control over the operation of interconnected transmission facilities within its region.

- An ISO should identify constraints on the system and be able to take operational actions to relieve those constraints within the trading rules established by the governing body. These rules should promote efficient trading.
- An ISO should have appropriate incentives for efficient management and administration and should procure the services needed for such management and administration in an open competitive market.
- An ISO's transmission and ancillary services pricing policies should promote the efficient use of and an investment in generation, transmission, and consumption. An ISO or an Regional Transmission Group (RTG) of which an ISO is a member should conduct such studies as may be necessary to identify operational problems or appropriate expansions.
- An ISO should make transmission system information publicly available on a timely basis via an Open Access Same Time Information System (OASIS)
- An ISO should develop mechanisms to coordinate with neighboring control areas.
- An ISO should establish an Alternative Dispute Resolution (ADR) process to resolve disputes in the first instance.

From FERC's guidelines for ISOs, it follows that ISOs should create efficient transmission pricing. In general, transmission cost is only 2% of a major investor-owned utility operating cost. However, transmission pricing is important because they provide price signals that can create efficiencies in the power generation market. Well defined transmission prices can influence the investment in new transmission capacity, load and new generation location. FERC has recognized the importance of the transmission pricing principles which complement the ISO guideline principles [3]:

- Transmission pricing must meet the traditional revenue requirements of the transmission owners.
- Transmission pricing must reflect comparability. Comparability means that a transmission owner should charge itself on the same basis that it charges others for the same service.
- Transmission pricing should promote economic efficiency.

- Transmission pricing should promote fairness.
- Transmission pricing should be practical.

ISOs face a number of new problems through the transmission pricing, one of which is how transmission pricing can be determined on a fair and realistic basis, reflecting the physical set–up and conditions, including power flow loops and multiple paths. Note that FERC suggests that "Transmission pricing must meet the traditional revenue requirements of the transmission owners", that it should be fair and practical. This implies that the traditional methods had to be fair and practical. If this is true, the question is, why do we need new transmission pricing? On the other hand, if it is not true, it is very unlikely that the ISO can make it true and at the same time satisfy the first principle. Another problem is congestion control through transmission pricing. It is important to note that the transmission pricing sends signals to market forces which can easily misunderstand them.

In addition to the ISO idea, the newly developed market must provide a new way of power trading. The two main approaches are bilateral trading and power exchange trading. Bilateral trading is based on buyers and sellers working out the prices and terms of the contract themselves. The power exchange pool is based on the buyers and sellers submitting the offers and the demand to a control office, and the purchase contracts are done through this entity. There are a few power exchanges (PX) in operation, in spite of many market participants claiming that they are unnecessary and that the same efficiency and functionality can be achieved through the bilateral market. A

small technical complication for the ISOs is that the PX could be a part of the ISO or a separate entity. In either case, FERC requires ISOs to monitor the market, identify abuses by the participants and take corrective actions. This requires very close coordination between the ISO and PX. If a PX is under ISO control or part of it, a very close coordination can be achieved. However if the PX is part of the ISO, there could be conflict of interest situations, depending on other factors.

ISO operation is highly dependent on the information provided by different utilities. FERC requested a real-time, on-line information system to be available to transmission providers and transmission customers, containing information on transmission capacity reservations, ancillary services and transmission prices. To achieve these goals, an Open Access Same-Time Information System (OASIS) is in the process of development and testing. OASIS is an Internet based service, available to all power market participants, providing the information described previously. OASIS is being developed in two phases. It became operational in January of 1997. The second phase is the testing phase and FERC, as well as the industry, will make change requests at its end. OASIS participation is mandatory for the utilities under FERC's jurisdiction, which is, according to [3], about 70% of the nation's transmission grid. At the time of this writing, OASIS is still a new tool being developed by the industry and with a very uncertain future.

During the above discussion, a deregulation approach was described, based on the direction in which the industry is going. This approach depends heavily on a complete unbundling of the generation, transmission, and transmission control. The most critical

point from both technical and economical points of view is the role of the Independent System Operator. In technical terms, the ISO role is close to an area controller's in the vertically integrated market. However, with the new requirements introduced with deregulation, the ISO's role is much more complicated. This is because of fairness requirements as well as limited information about the behavior of the market forces and sometimes limited control over both generation facilities and transmission owners. New communication, analysis, control, and optimization tools are needed to cope with the additional complexity. These would include new telecommunication tools, new market analysis tools, and modified or completely new optimization tools. All of these tools will be interdependent and of equal importance in a final, operational system, but at the moment, it seems that the revision of the old optimization tools and their applicability in the new environment is of the highest importance.

1.2 Software and Hardware Tools Required for Free Market Implementation

As mentioned previously, there are different scenarios applicable to free market implementation. It is almost certain that the industry is opting for the so-called "maximum ISO scenario". In the maximum ISO scenario, the ISO has the maximum control of the power system operations, as well as the maximal amount of information provided by all the participants to perform its duties. All the participants submit bid curves for supply with appropriate physical constraints and demand for various time frames. A global unit

commitment is solved by the ISO, and based on the provided information and demands, the unit commitment decision is done one day ahead. However, in real time the ISO might choose a different dispatch for a number of reasons. The ISO also sets the modal prices, including the transmission congestion component. Since the ISO is also responsible for system operation and security, in addition to system economy, the transactions will only take place if the security constraints are not violated. To provide adequate service, the ISO needs appropriate tools for transmission planning, reliability analysis, economic dispatch and unit commitment, system control, security contingency analysis, performance evaluation, costing, risk management and forecasting, power quality and protection.

Transmission planning is a very sensitive issue. Existing tools for load forecasting must be modified and extended to account for new problems. Two of the main questions are who is responsible for the transmission infrastructure planning and development, and who is responsible for detecting a need for additional transmission capacity. High congestion might mean higher revenues for some generators and limited access for others. In addition to tools capable of detecting such conflicts, an ISO will also need market simulation tools to analyze the impact of new transmission investments on the rest of the market. Both technical and market analysis must be done carefully so that the transmission modifications and change in price do not result in unnecessary expense for the public.

Reliability is an ancillary service, and as such, its expense was never really deter-

mined. However, in the free market economy, it will be necessary to estimate closely the cost of almost all services provided by an ISO including the reliability. This will require new (at present, non- existing) tools to make such an estimate. Also, the industry participants must agree upon what an acceptable level of reliability is, and how much it is worth to every party involved.

Unit Commitment (UC) and Economic Dispatch (ED) are old tasks which must be revised for the new market environment. UC and ED are not the same in the deregulated and the regulated power market. Utilities in the regulated market solve global UC and ED for every generation unit based on the present and forecasted load. In the deregulated market, both UC and ED are viewed from two different perspectives, that of the Independent Power Producers (IPPs) and from the perspective of the ISO. An IPP has the goal to minimize its expenses and maximize its profits. An ISO is concerned with the stability of the system and minimizing the total cost of the presently needed power. There are different scenarios with respect to ISO involvement in IPPs' operations. One possibility is that the ISO takes into account all the constraints of every generating unit and develops needed UC and ED. This is almost equivalent to the traditional non–deregulated approach. Another possibility is that the ISO collects available bidding curves from all the producers and develops the UC and ED schedule over a period of time. This is similar to the traditional approach but without the consideration for units' constraints and replacing the traditional power production curves with the producers' bidding curves. Although there are similarities between both approaches and

the traditional OPF, there are also significant differences that could influence the final power cost. To illustrate the complexity of the new requirements, suppose that an IPP offers power at the lowest possible price. However, its units might be on the ISO's "must not run" list. In the new market, this IPP should be compensated somehow even though it cannot sell its power and will not incur any expenses. On the other hand, an IPP which is on an ISO's "must run" list might charge almost any price. Also, small distributed generators are becoming a significant factor in providing stable, high availability power. It is clear that the ISO should come up with the best power production and delivery at the lowest price in spite of the IPPs' attempts to maximize their profits. This is a new situation which cannot be addressed with classical optimization tools due to new multiobjective goals under new, ill–defined constraints.

System control is low level control including generation rescheduling, voltage adjustments, capacitor switching, etc. and is also the ISO's responsibility. New tools are needed to control power flows using Flexible AC Transmission System (FACTS) devices in an ISO centralized manner. Real–time measurements require real–time communication facilities for fast control of system dynamics. Area controllers had the same problem, but now the problem is more complex, due to the increased size of the controlled system and additional economic issues.

These are just a few of more important problems faced by the free market that need new or modified tools. For this particular scenario, two types of tools can be identified. Telecommunications and information publishing is a requirement by FERC.

It is clear that the databases accessible by all the parties involved would be of significant size, requiring reliable software tools for managing them, as well as reliable and fast communication channels for their access. The second group of tools required are flexible optimization techniques. These techniques must be able to solve constrained, highly nonlinear, discontinuous and non convex optimization problems. This work is concerned with the latter group of problems.

1.3 Dissertation Objective

The dissertation objective is to apply Artificial Intelligence (AI) methods to the deregulated power system problems and demonstrate the ability of AI methods to successfully address these issues. The AI solutions are compared with existing solutions if available. The AI methods are used on problems previously not solved by classical methods as well as some classical problems from a different perspective and a slightly different goal.

From the previous discussion of the US power industry, we recognize the importance of optimization tools in an ISO everyday operations. Unit commitment and economic dispatch are perhaps the most important problems in both the regulated and deregulated power market having side effects on the entire power system operations. There are different approaches to the economic dispatch, and it is usually implemented using optimal power flow. Therefore, in this research, we consider the optimal power flow and the unit commitment (UC) problem in the new deregulated market. Due to the unbundling of power system services under deregulation, UC and economic dispatch

(ED) are considered separately by each unbundled entity as trying to maximize its profits or provide reliable operation in case of the ISO. Efficient UC and ED are of interest to both power producers and ISOs but with different almost complementary considerations. In this work we address the UC and ED from the point of view of an ISO. Power producer considerations are addressed elsewhere [5] without consideration for an ISO's concerns.

There are two AI methods used in this dissertation. The first method is the Genetic Algorithm (GA) optimization and the second method is the Decision Tree (DT) machine learning. The Genetic Algorithm optimization is a stochastic type of optimization based on the selection mechanism found in nature where the "fittest individual" dominates the population and survives. The fittest individual is found through the stochastic mutation and crossover process. Decision trees are a form of machine learning from examples. They can be used for classification and regression. There are different types of decision trees with different training algorithms. They can produce 100% classification accuracy on the training set but can also be simplified (by pruning them) and lose some of their accuracy or smooth out the noise in the training set. We use unpruned DTs for our purposes.

Classical ED and UC methods are restricted in various ways depending on the underlying algorithm. In this work, we will use the Optimal Power Flow (OPF) as the basis for both ED and UC. The goal is to develop a flexible OPF which can deal with problems impossible or difficult to solve with classical methods. Such an OPF solution

can be used as a building block for a better ED and UC. The main problems that we will try to solve with the AI approach are the flexibility of the objective function, ease of defining the problem, real time operation, and load forecasting uncertainties. The objective function and its accompanying constraints are difficult to translate into formal mathematical terms needed for a classical optimization method, and sometimes such a transformation is not even possible. Since the mathematical formulation of the power system state solution is a non–linear multidimensional function, a non–linear optimization technique must be used to locate the global optimal point. Constrained non–linear programming is a difficult problem, and in the case of power systems, it is of a questionable success. In addition to its nonlinearity, the objective function could be discontinuous and non–convex. Stochastic AI optimization techniques are usually successful when applied on problems with these characteristics. Using AI methods it is possible to have good results with almost any type objective function. The only restriction is that the objective function is a single valued function, but that does not prevent us from having multiple goals. It has been reported that even the existing OPF solutions are rarely used due to their hard to understand user interface, large amounts of input and output data, and many computing parameters that must be adjusted for different cases. An algorithm easy to use in research and production environment is another goal of this dissertation. There is not much benefit from a tool that no one knows how to use. Although a power system operation could benefit from an AI approach, it is known that AI methods are computationally demanding. We will address

possible ways of achieving acceptable on–line operation in the case of UC. The last area that will be investigated in this research is how to handle load uncertainties in a power system network. This type of uncertainty was chosen because of the high likelihood of substation distributed generation being deployed within a power system. Distributed generation will likely be located within the distribution system invalidating at least partially, any load forecasting efforts done by the ISO in the case of a deregulated market, or a utility in the case of a vertically integrated market. The distributed generation can be considered as a part of the load causing a higher or lower level of load uncertainty. Currently, these types of problems are handled with statistics and probability theory. Here, we will take a different route. Instead of pursuing a better estimate of the load, we will try to estimate how close a present unit commitment is to being invalid based on the real–time measurements. This will give a human dispatcher an idea what kind of changes need to be made and how soon. This will be done using Decision Tree (DT) machine learning.

The main contribution of this work depends on the point of view of the reader. A programmer might find the flexible data structures and routines used to calculate the power systems state to be the main contribution. An AI researcher could argue that the working real–life AI application is a great proof of AI methods maturity. A mathematician might find interesting that some problems unsolvable with classical methods are possible to solve with stochastic optimization approach. For a power engineer, the main contribution of this work could be a tool that can be used to easily analyze new

scenarios arising in the deregulated power system environment. We believe that in addition to all the different uses of this work, it has a value as a good starting point for future research. There are many directions this research can take. The decision tree use in power systems could be particularly interesting.

This dissertation incorporates a few different areas from power systems theory and AI theory. This is not a research in new AI algorithms. This is a research in AI applicability to hard to solve power system optimization problems. Any proven AI algorithms and resources found useful are used to develop stable and reliable power system solutions. Classical power system algorithms are reviewed in the appendices. AI material is introduced in a general framework, then discussed in general power system terms, and finally applied to OPF, UC, or both. The Power Flow theory is reviewed in Appendix A. OPF formulation and the Sequential Quadratic Programming solution is described in Appendix B. The theory and complexity of UC and some classical solutions are reviewed in Appendix C. All the program listings can be found in Appendix D.

Chapter 2

Genetic Algorithm Optimization

The genetic algorithm is a stochastic optimization method based on simplified Darwinian evolutionary theory. During the last forty years, a number of different genetic algorithms were developed including, but not limited to, genetic algorithms, genetic programming, evolutionary strategies, etc. The term genetic algorithm evolved into evolutionary computations used to refer to all algorithms with genetic algorithm origin [22]. In addition to these derived evolutionary computational methods, there is a number of different hybrid algorithms incorporating evolutionary computations and optimization methods from other scientific areas such as fuzzy logic and neural networks.

2.1 Generic Genetic Algorithms

A general Genetic Algorithm (GA) has five basic components: a genetic representation of a problem's solution, a solution population initialization method, a population and

individual fitness function, reproduction functions, and genetic algorithm parameters.

Genetic representation of the solution can be the same type of variable as the solution itself or there can be intermediate mapping between the two spaces. The original genetic algorithm [12] closely mimics the natural genetics model by using binary strings to represent the genes or equivalently the solution. For the problems with a real variable solution space, mapping is done between the binary strings and real variables. In addition to more complex computations, this introduces theoretical difficulties as well. For example, the Hamming distance between 10000000 and 01111111 is maximal while the Euclidean distance is minimal. This means that to make a minimal change in the solution variable, all the bits in the gene must be changed simultaneously, effectively introducing a discontinuity in the search space. It is much more reasonable to have real variable genes having one–to–one mapping with the solution and avoid any unnecessary mapping. This requires a different crossover and mutation approach removing the genetic algorithm even further from its origins in nature. These changes make the GA efficient, accurate and usable for optimization of engineering systems with a large number of variables. With respect to the type of allele encoding, the encoding methods are classified as binary encoding, real variable encoding, integer or literal permutation encoding, and general data structure encoding. Real variable encoding is suitable for constrained or unconstrained function optimization since it is superior to any other encoding for these types of problems.

A GA starts its evolution from an initial population of possible solutions and moves

toward an optimal point based on a selection process. The initial population is chosen randomly, and its individuals must be within the constraints defined by the problem. An individual must represent both a feasible and legal solution of the underlying physical problem. This means that it must be a physically feasible solution and within the variable constraints. If the alleles are binary strings, the initial population is a binary string where each bit is 0 or 1 with 0.5 probability. If the alleles are real variables, the initial value of an allele is a random number between the minimum and maximum limits of the variable represented by the allele.

The population and individual fitness function represents optimality of the solution and is the function that is being either minimized or maximized. The fitness function used by the GA does not have to be an objective function based on the physical performance of the system. The fitness function can be in some way a scaled version of the objective function. If the objective function is a non-negative real function, and the alleles are real variables, it is not necessary to use any scaling. The fitness function depends on the problem and what needs to be optimized.

Reproduction functions are chosen to efficiently perform both exploration and exploitation. There are different mutation and mating schemes that can be used. The reproduction functions are inspired by the natural genetic principles and genetic algorithms include following operators:

- selection,
- recombination or crossover, and
- mutation.

Selection is responsible for for maintaining high quality solutions to be propagated to the next generation of solutions. Recombination is used to form a new solution from two good solutions. Mutation provides randomness to the search by allowing alteration in possible solutions. These three operators are sufficient to provide an adequate degree of exploration and exploitation and allow the genetic algorithm to quickly converge sufficiently close to a global minimum within the allele constraints.

The GA parameters control process termination, mutation probability, crossover procedure, and fitness function scaling. The termination condition is usually the maximum number of iterations or a maximum change in the optimal point found in recent populations. Either terminating condition is acceptable and performs well. Mutation probability is just the probability of introducing mutations in genetic material and is a value between zero and one. The crossover procedure is a procedure of mixing the genetic material between two parents. This is highly dependent on the allele types and can be bit flipping in the case of bit strings, or changing the real variable allele according to some distribution in the case of real variable alleles. Fitness function scaling depends on the range of fitness values and the crossover procedure. Non–negative fitness functions usually do not need any scaling. Figure 2.1 shows GA steps.

2.2 Genetic Algorithms Applied to Power Systems

Variations of the genetic algorithm have been applied with various degrees of success to different power system areas including planning, operation, and analysis. Scheduling

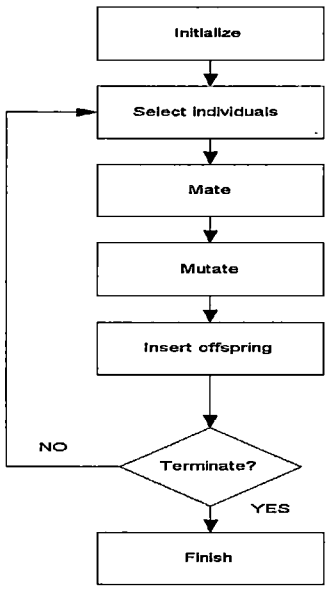


Figure 2.1: Genetic algorithm steps.

maintenance of electrical power transmission networks using genetic programming was reported in [34]. Environmentally constrained economic dispatch was addressed in [18]. Binary string GA was applied to the unit commitment problem in [2]. GA was also employed to solve economic dispatch of a system with cubic cost function units [8]. An interesting application of genetic algorithms to multiple power flow solutions was reported in [38]. Some of these applications are focused on improving the computational speed of the GA algorithm, often at the expense of the accuracy, not realizing that classical binary string GA is not suitable for these types of problems. Here, we will introduce a flexible GA framework implementing both binary strings and real variable alleles. When used properly, the GA can produce good results in acceptable time. We opt for a fully tested GA implementation with enough features to cover areas of interest

and consistent programming interface.

A GA is a fairly simple algorithm and there are many implementations in the form of programming libraries and full applications. Neither a simple library nor a complete application has enough flexibility needed for use in a real life applications but they are usually tested and have stable implementations. In this work, we use GALIB based on [33]. GALIB is a Genetic Algorithms library written in C++ providing the flexibility needed to implement new algorithms, replace any part of the existing algorithm, or even paralellize it. The existing GALIB functionality is used where appropriate or replaced with customized functions when needed. The rest of this section describes GALIB features in general terms relevant to Power System operations.

GALIB can be used with different allele types including floating point, integer, binary string, or any user defined type which provides appropriate operators. A set of alleles are stored in one, two, or three dimensional arrays representing genes. What type of array is used depends on the problem that is being solved. For example, a traditional UC would use a two dimensional array of binary strings to capture the temporal dimension of the problem. An OPF would use a one dimensional array of floating point numbers representing system's controls at a single point in time. Genes of a population represent the control variables of a power system and as such are real variables. Depending on what the optimization goal is, this could include generator outputs, voltage levels, transformer settings, and shunt capacitor values. The value of a control variable is the value of the corresponding allele. There is no mapping

between the solution and representation space making constraint enforcement trivial. Alleles can be defined as continuous variables, discretized continuous variables with a fixed discretization step, or as a set of possible values. The ability to have discretized control variables is important in power systems where some of the controls change in steps. The GA with discretized alleles also seems to be considerably faster. No classical optimization method with discrete variables has been applied to power systems scheduling problems.

The solution population is initialized with random values respecting the type of the alleles as well as the range. If the variable represented by an allele is discretized, the initial value will be from the set of discrete values and it will be between its minimum and maximum limits. It is possible that a random population would not represent a realistic power system state or that the Jacobian of the system would be singular. However, this would not pose a problem since this is just the initial population. The final solution would be feasible and have a legal solution if the system is physically feasible and described correctly.

GALIB implements a simple GA, a steady state GA, an incremental GA, and a deme GA. The simple GA uses non–overlapping populations and is described in [12]. The steady state GA uses overlapping populations with the overlap size defined by the programmer. Populations are evolved in parallel, and at the end of each evolution, predefined number of individuals are exchanged. The incremental GA is a population overlapping algorithm with one or two individuals. The deme GA has multiple inde-

pendent populations. The best individuals in a population are migrated into another population. The master population contains the best individuals from each population. The GALIB allows derivation of new GA algorithms using the inheritance property of Object Oriented Programming (OOP). Simple and steady state GA variations seem to give good exploration and exploitation properties for the power system problems. Whether we use a GALIB or our own algorithm, the solution must be both feasible and legal. Any solution found by a GA is both feasible and legal although not all solutions are optimal. The optimality of the solution is based on the fitness function closely related to the power system cost function. Since the cost function is non-negative, no complicated scaling is needed which allows the use of a power system cost function as a GA fitness function. Due to the flexibility of the GA, the cost function can be a multiobjective function. That is, optimization could be done in terms of different goals at the same time. Multiobjective optimization is a complex field and a GA can benefit from formal multiobjective research theoretical results [29] or the objective function can simply be a sum of weighted operational and penalty costs. It is important to make a distinction between control variable constraints and controlled variables constraints. Control variable constraints are easy to enforce while controlled variables must be constrained by using exponential or polynomial penalty functions. Bus voltages must be between predefined limits to satisfy voltage stability requirements. If the voltage exceeds the limits, a penalty is added to the cost function. The penalty function can be exponential or polynomial. Figure 2.2 shows a typical polynomial penalty function.

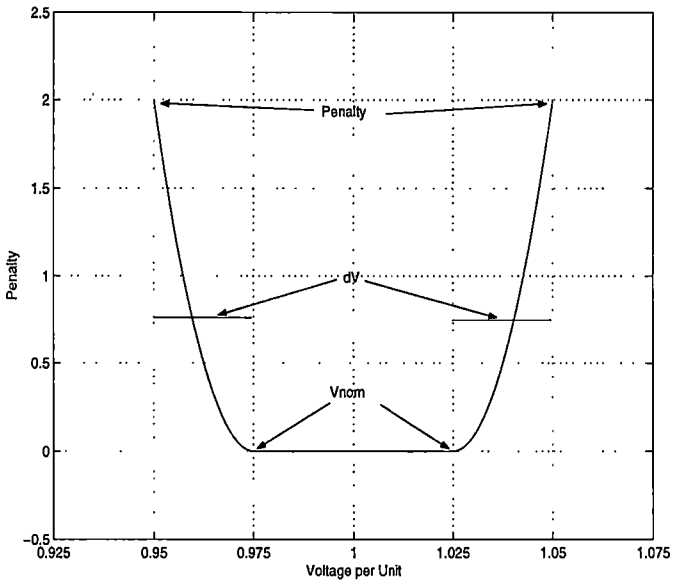


Figure 2.2: Voltage penalty function example.

Second order polynomial coefficients can be derived from the desired curve features using the following equation:

$$penalty(V) = A \times V^2 + B \times V + C \tag{2.1}$$

where

$$A = \frac{Penalty}{dV^2}$$

$$B = \frac{-2 \times Penalty \times Vnom}{dV^2}$$

$$C = \frac{Penalty \times Vnom^2}{dV^2}$$

GALIB provides means for different allele types and implementation of new ones through OOP. For power systems, we will use floating point alleles to represent the sys-

tem's control variables. Available selection operators of interest include rank selector, roulette wheel selector, tournament selector, and uniform selector. Rank selector selects the best individual. Roulette wheel selector picks an individual with high fitness score relative to the rest of the population. Each individual has a probability p where p is the ratio of the individual's fitness and sum of the fitnesses of the rest of the population. The higher probability p means the higher the probability of the individual being chosen. Tournament selector selects two individuals using roulette wheel selector and then picks the one with a higher fitness score. The uniform selector picks an individual randomly from its population. The roulette wheel and tournament selectors are appropriate selection schemes for the OPF and UC. Replacement operator replaces the worst individual which seems to be the best policy for power system problems.

Crossover between parents is done by randomly choosing a value between parents' values based on uniform distribution. Similarly, mutation is performed by applying a perturbation to an allele value based on Gaussian distribution function. Different probability distribution functions can be chosen or implemented through OOP.

GALIB parameters can be assigned from the program's command line, from a configuration file, or adaptively during the program execution. Parameters that can be assigned include the size of a population, number of generations, termination condition, mutation probability, the number of replacements, the number of offsprings, logging frequency, and variables to be logged. Parameters can be divided into two groups. The first group is the parameters that influence the GA accuracy and execution times and

they might need to be readjusted when applied to optimization problems of significantly different sizes. The second group of parameters define the execution logging and can be used to fine tune the accuracy and execution times of the entire procedure. Not all of the parameters are applicable to all of the GA algorithms.

Chapter 3

Genetic Algorithm Based Optimal Power Flow and Unit Commitment

The ultimate goal of power systems operations is maintaining reliable service (keeping the lights on), while the production cost is optimized. This formulation is general and includes different formulations of system reliability and cost optimization. The optimization procedure is significantly complicated with the introduction of deregulation. Economic Dispatch (ED) and Unit Commitment (UC) are the basis of the steady state operations. There is a number of different classical ED and UC solutions. While ED and UC are discussed separately in textbooks, they are essentially two parts of a single optimization problem. In this work, ED will be addressed as a part of UC and based on

optimal power flow which allows one to optimize different quantities in power system operations. Although deregulation introduces additional constraints and requirements, it also separates ED and UC into the resource owner's and ISO's side of the problem. This dissertation recognizes the ISO's problem as more important and considers its point of view rather than the generator or transmission owner's problem. In general, an ISO is not concerned with many of constraints which were a part of classical ED and UC. For example, an ISO does not need to account for constraints such as up and down times of a generator. However, it must consider new issues such as spot market prices, and transmission congestion control. Classical ED and UC are not suitable to address these types of issues. In this chapter, a GA will be applied to a classical formulation of the OPF and UC and the results will be compared with classical results. A GA will also be applied to a hypothetical situation in a deregulated market but for this situation there is no benchmarking alternative.

3.1 Power Flow Implementation

The Genetic Algorithm OPF is based on an objective function which uses a power flow to calculate power system state after a new set of control variables are applied by the genetic algorithm. The power flow used is the familiar Newton–Raphson non–linear equation solver. Appendix A reviews the mathematics behind the power flow calculations. The power flow implementation is based on power flow description and discussion found in [37, 36, 6]. There are different levels of modeling complexity that can

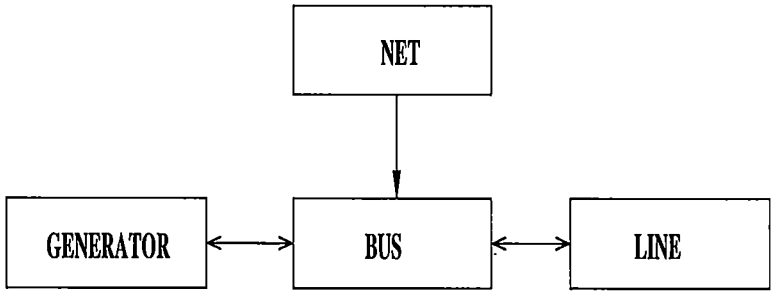


Figure 3.1: Power flow data structure diagram.

be accounted for in a power flow algorithm. This implementation allows MW and MVar limits for each generator. If a generator exceeds the MVar limit, the MVar generation is set at the limit and the bus voltage is not constant anymore. When this happens, the structure and the calculations of the Jacobian are changed due to non–constant voltage. Real power cost function is defined as a second order polynomial. Transformer settings are continuous variables and if discrete step changes are desirable, they must be assigned in discrete steps. Shunt capacitors are implemented and can be used as control variables.

The programming interface is done in OOP manner similar to the one reported in [17]. Figure 3.1 shows the relationship between different objects. These are not derived classes, and there is no run time virtual function resolution overhead. Second level objects are just encapsulated in the Net class. From the diagram, each bus knows what generator, load, shunt capacitors, and lines are connected to it as well as what are the neighboring buses. The generator has the information about what bus it is connected to. Branches carry information about the impedance, whether it is a transformer or not

and if so, which end is which side of the transformer, and the transformer setting. Any component of the network is easily accessible and modifiable. Most of the interfaces between the parts are implemented through pointers so that there is no information duplication overhead. The small memory overhead is well justified by the time saved by not searching for a component through complex data structures. The Net class encapsulates all the components allowing for multiple Net objects. This was done for future development which could include multiple areas and calculation of power wheeling between neighboring areas.

The power flow solution starts by building the admittance matrix of the system, assigning the initial solution, and calculating the initial Jacobian matrix. Since power systems are large, sparsely connected systems, sparse matrix methods are used for the admittance matrix and Jacobian. This requires using special sparse matrix linear equation solvers. Sparse matrix structures are encapsulated in a class. Elements are stored in an array of Standard Template Library (STL) [16] maps. The array index is the column index, index into the map is the row index and the map value is the matrix entry at the row/column location. Standard template library maps are implemented as binary trees and all the operations that need to be performed on this data structure are very fast. The maximum time needed to access an element is proportional to $log_2 n$ where n is the size of the matrix. A linear search would take on average time proportional to $\frac{n}{2}$. Note that in the first case we estimated the maximum time, while in the second case it is average time. For example, to access an entry in a matrix of

$1,048,576 * 1,048,576$ dimensions would take at most 20 comparisons while the linear search would take 524,288 comparisons on average. Traversing the above described structure produces the exact sequence that we need for the standard Harwell–Boeing format as described in [13]. At this point, the sparse matrix solver can be any solver available which uses standard column compressed matrices. Switching between multiple solvers is also possible since the solver is a class method that can be overloaded. We had good results with both a public domain solver (SuperLU) [15] and a commercial solver (IMSL) [32]. A nice feature of SuperLU is that the ordering does not have to be redone if the matrix format did not change. The latest (version 4.0) IMSL version has a memory leak for which the company e–mailed a temporary solution. IMSL was used because of the convenience of having other library routines available. The admittance matrix does not have the Harwell–Boeing representation since it is not involved in any equation solution. It is used purely as a fast access container. The Jacobian matrix does need Harwell–Boeing representation and it is recalculated at each solution iteration and if needed, is reformatted. After the solution is obtained, line flows, losses, and charging Vars are calculated and are available for output or for calculating the objective function. Any of the output variables can be used for the objective function calculations and any system settings, generator outputs, voltage levels, and such are available as control variables to be applied through the genetic algorithm.

3.2 Genetic Algorithm Based Optimal Power Flow

The OPF is a very important tool in system planning and operations. Optimal Power Flow, as its name suggests, solves the power flow problem and schedules the network controls in a way to minimize an objective function. Since its original formulation, the OPF algorithm has been continuously faced with more and more challenging problems. Frequently, it needs to include security constraints and it is desirable to consider market and even environmental issues as reported in [18]. With deregulation, it was clear [30] that the OPF will have an even more important role in future energy markets. The OPF is presently inclùded in a majority of Energy Management Systems (EMS) but they are rarely used and the main reasons for this are [9]:

- Insufficient training
- Inadequate Man Machine Interface
- Very large amount of input data
- Large number of tuning parameters
- Pages and pages of displays for the inputs and outputs
- Output too difficult to comprehend except for an OPF expert
- Difficult to debug

However, the OPF needs to be applied to a number of problems such as [9]:

- Security Constrained Economic Dispatch (SCED)
- MW and MVAR loss minimization
- Constraint Priority Optimization
- Hierarchical control
- Remedial actions or corrective rescheduling

- Preventive scheduling or contingency constrained dispatch
- Maximum transfers capacity
- Closed loop control
- Capacitor and reactor placement
- Switching action
- MVAR cost minimization
- Voltage control
- Sensitivity analysis
- Transaction evaluation
- Dispatcher training system scenario building
- Wheeling loss computation
- Nodal price for power
- Environmental dispatch
- Constraint costing

Each of these applications has specific optimization requirements and different ways of interfacing with the rest of the network. For example, in the case of SCED, the dispatch is typically run every fifteen minutes while the AGC is run every two seconds using the set points from SCED. If the OPF is used to minimize MW and MVAR losses, this can result in substantial savings. In general, there are a number of constrained variables that might be desirable to be included in the OPF, but if the number of constraints exceeds the number of controls, there might not be a feasible solution. In that case, the dispatcher can assign weights to the violated constraints and try to minimize the weighted sum of violations squared. The OPF can also be used for corrective

rescheduling or remedial actions. Preventive scheduling and contingency constrained dispatch can also benefit from OPF as well as many constrained optimization problems.

The OPF is being used for many diverse problems in power systems, however it cannot address all the problems at once. With deregulation, reliable solutions to all the old problems are even more important since the economic constraints are likely to lower the safety margins. A basic formulation and sequential programming solution is reviewed in Appendix B.

A Genetic Algorithm OPF (GAOPF) can address the classical OPF problems and extend its applicability and usability. A GAOPF can employ more flexible objective functions. A GAOPF can successfully deal with objective function nonlinearities and find the global optimal point. It is also much easier to define an objective function for a GAOPF than for a classical OPF. All these benefits are significant in an environment where the tools change frequently and the development and testing times are extensive. The deregulated power systems market is certainly such an environment.

The Genetic Algorithm (GA) has been applied to a numerous optimization problems. Usually, a GA is tried when all other possibilities are exhausted due to the complexity of the optimization problem or computational inefficiency of available methods. A GA is more likely to produce a global optimal point than classical methods. In this chapter we will discuss benefits and drawbacks of using a GA implementation of the OPF. Results will be compared with results obtained with other OPF packages using a general purpose Sequential Quadratic Programming (SQP) described in Appendix B.

Genetic algorithms used for OPF are based on [33] and reviewed in Chapter 2. The power flow algorithm is a part of the GAOPF and is described in Appendix A.

3.3 GAOPF Operational Principles

A Genetic Algorithm OPF has the same objectives as the classical OPF but the approach is more flexible allowing for easy incorporation of different control variables and goals. The possible control variables include generator outputs, transformer settings, shunt capacitor settings, etc. The goals could be minimizing production cost, minimizing losses, or minimizing transmission congestion, among others. All of the controls are represented with floating point alleles limited by their minimum and maximum values. To be able to use all of these controls, there must be fitness function parameters and the fitness function must have a way of applying these controls to the simulated power system. Under these conditions, the GAOPF is capable of evolving from an initial solution to the global optimal point.

An initial solution is a random operating point satisfying predefined constraints. The initial solution consists of a set of alleles representing controls applied to the system. The alleles themselves can be discretized or continuous. In the following examples, generator output levels will be controlled using continuous alleles to be able to compare the results with the classical OPF results. The program also has command line options to use discretized controls and discretized controls with zero included. When an allele representing a generator output is zero, it is effectively like turning that generator off.

This would allow an easy solution for the unit commitment problem. However, an isolated value is hard to reach during the GA exploration. The performance of the GA with such alleles is rather unsatisfactory.

The fitness function is evaluated from system state after controls are applied. The state of the system is found from the power flow calculations. Power flow is coded in an OOP manner using sparse matrix techniques. The entire system is represented as a single object consisting of subobjects representing buses, lines, generators, and transformers. All the components of the system are accessible through message passing methods. This means that anything can be easily considered as a control variable. In addition to common production and loss costs, penalties are added for exceeding the limits of controlled variables. For this, a second order polynomial function is used as described in Chapter 2.

Reproduction functions used to search for the optimal point are as described in Chapter 2. The probability distribution function used for crossover and mutation is the familiar Gaussian distribution.

3.4 GAOPF Examples

The following examples illustrate a GA implementation of the OPF. Different objective functions will be discussed and compared when appropriate with Power World's OPF [35] or the generic Sequential Quadratic Programming (SQP) approach [32]. Power World is a commercial package originally developed at The University of Illinois [35]

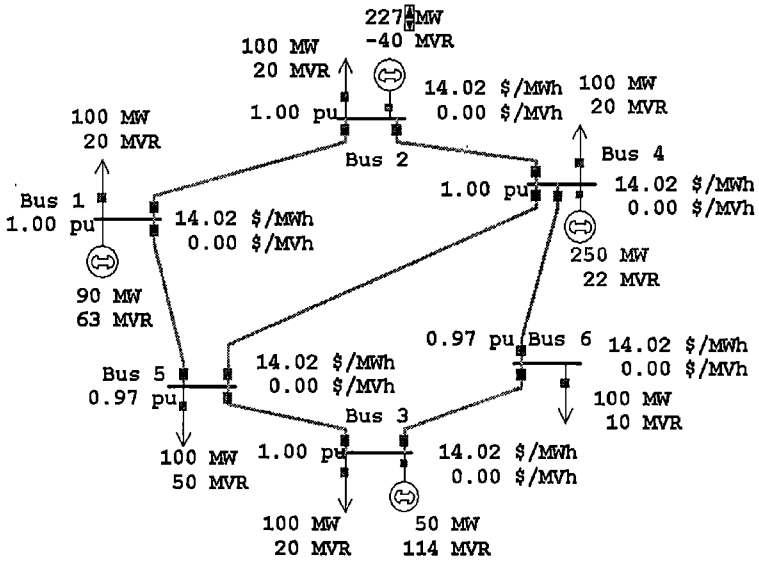


Figure 3.2: Power World 7.0 output.

by the faculty members and students. GAOPF results are compared with two versions of Power World, version 5.0 and version 7.0. Single goal and multiple goal objective functions will be compared and different possibilities in the deregulated power systems will be discussed. For illustration and comparison purposes, small example systems will be used and at the end, performance issues and possible solutions will considered. Similar results were obtained with the systems up to 513 buses and 67 generators. Figure 3.2 shows the Power Flow diagram of the power system and Table 3.1 lists the details not displayed in the diagram. All the lines have a resistance of 0.04, a reactance of 0.08, and a charging capacitance of 0.02 per unit.

Table 3.1: GAOPF Optimization of the production cost.

Gen No	MW Out	Volt	MVar Max	MVar Min	MW Min	MW Max	Cost Func
1	0.91	1.0	10.0	-10.0	10	250	$105.0 + 12.0 * P + 0.012 * P^2$
2	1.96	1.0	10.0	-10.0	10	250	$96.0 + 9.6 * P + 0.0096 * P^2$
3	0.86	1.0	10.0	-10.0	10	250	$105.0 + 13.0 * P + 0.013 * P^2$
4	2.37	1.0	10.0	-10.0	10	250	$94.0 + 9.4 * P + 0.0094 * P^2$

3.4.1 Power Cost Minimizing OPF

The only optimization criterion in this case is minimizing the power cost while maintaining acceptable voltage levels:

$$Cost = \sum_{i=1}^{all\ generators} \alpha_i + \beta_i * P_i + \gamma_i * P_i^2 \tag{3.1}$$

subject to bus voltage limits:

$$V_{i_{min}} \leq |V_i| \leq V_{i_{max}}, \qquad \forall\, Buses$$

Power World 7.0 calculated the optimal point using the Linear Programming (LP) method resulting in a total cost of $7868.40 per hour. GAOPF resulted in a slightly lower cost of $7824.60 per hour. Power World 5.0 using the Newton–Raphson method arrived at the same result as GAOPF. The sequential programming method failed to converge to a solution close to either the GAOPF or Power World solution. None of the methods enforced the line flow limits. In addition, the Power World 7.0 result had exactly the same incremental cost for all the generators resembling λ–iteration Economic Dispatch method. The GAOPF incremental costs were close but not the same. The GAOPF computational time was considerably longer, but acceptable, taking almost

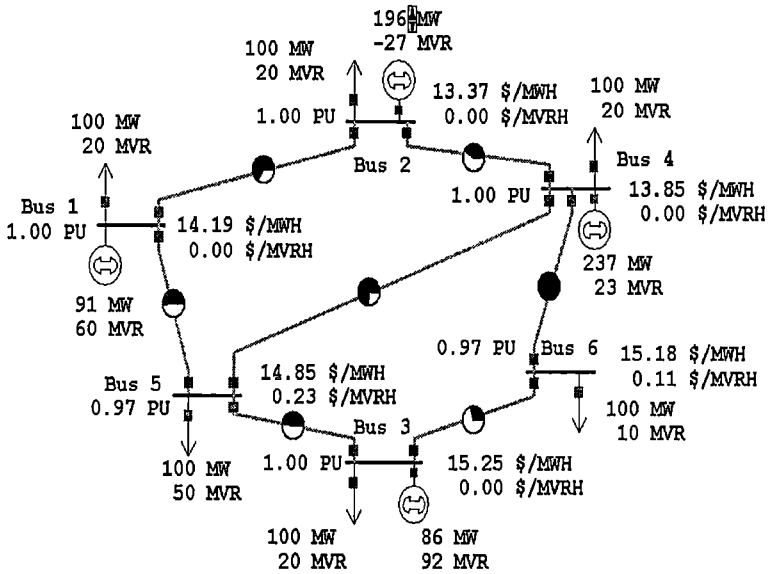


Figure 3.3: Power World 5.0 output.

one second. Due to the stochastic nature of the GAOPF algorithm, there was a small difference in the resulting cost on an order of $1.00 per hour between the runs. Figure 3.4 shows the convergence rate of the GAOPF. A number of larger networks were tested with similar convergence properties. Larger networks required considerably longer times to minimize the power production cost. A network with 513 buses and 63 generators required 12 minutes to find the optimal operating point. Long processing times are certainly of great concern and will be addressed appropriately. Sequential Quadratic Programming was consistently able to converge to the lowest observed cost when the GAOPF optimal point was used as the starting point. If higher accuracy is desired, SQP can be used to refine a solution found by the GAOPF. Newton–Raphson method from Power World 5.0 and GAOPF had very close solutions and almost exactly the

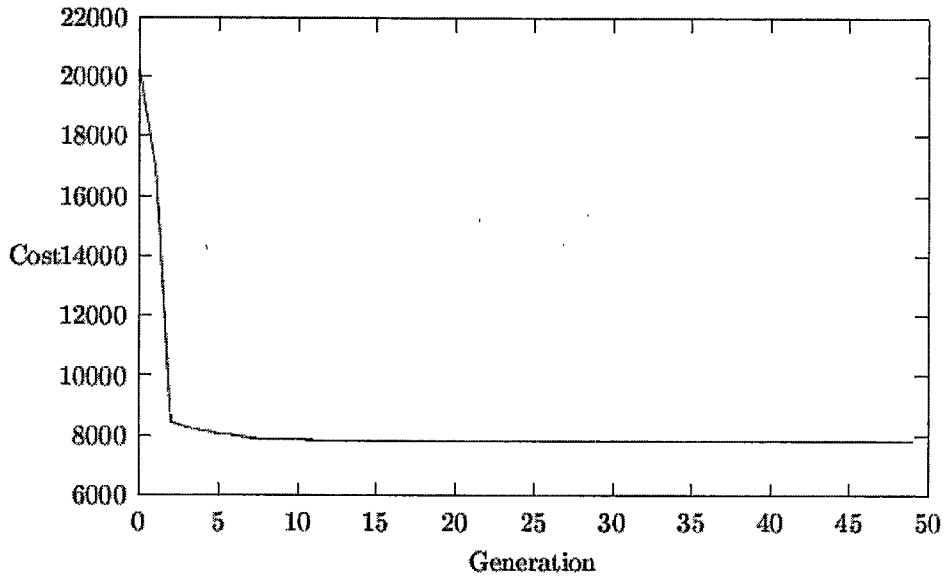


Figure 3.4: Genetic Algorithm Convergence Rate.

the same solutions after GAOPF was refined using SQP. GAOPF solution is shown in the table 3.2, while the Power World 7.0 and Power World 5.0 solutions are shown in Figures 3.2 and 3.3 respectively. The SQP solution, refined from the GAOPF solution, is shown in Table 3.3.

3.4.2 Power Losses Minimization

In this example, power losses are minimized while voltages are kept close to nominal voltage using the penalty function described previously:

$$Cost = \sum_{i=1}^{all\ lines} I_i^2 R \tag{3.2}$$

subject to bus voltage limits:

$$v_{i_{min}} \leq |V_i| \leq V_{i_{max}}, \qquad \forall\, Buses$$

The same power system is used as in the previous example but now, the GAOPF is compared with the SQP algorithm only. Tables 3.4 and 3.5 show respectively the GAOPF and the SQP results.

Both the GAOPF and the SQP result in the same value of the objective function, 3.8MW of loss but with a slight difference in the generator output levels and power factors. Voltage magnitudes are the same in both cases. It is clear that both algorithms arrived at the same result with respect to the objective function in common. The SQP always finished with the unique result which is to be expected since it is a deterministic algorithm. The GAOPF resulted in almost exactly the same value for the objective function between the runs but that is not necessarily true for other system variables. The production cost was significantly different between runs, demonstrating the fact that minimizing the production cost is not the same as minimizing the power losses. Even the SQP algorithm resulted in a higher production cost ($8187.65/hr) in this example as compared to the example in the previous section. The GAOPF had similar convergence characteristics as the GAOPF minimizing the production cost.

3.4.3 Minimizing the Production Cost and Power Losses

Production cost and power losses can be minimized at the same time by scaling one of the costs relative to the other one:

$$Cost = \sum_{i=1}^{all\ generators} \alpha_i + \beta_i * P_i + \gamma_i * P_i^2 + W_l * \sum_{i=1}^{all\ lines} I_i^2 R \tag{3.3}$$

subject to:

Table 3.2: GAOPF Optimization of the production cost.

From Bus	Volt. Mag.	Volt. Angle	Mw Load	Mvar Load	Mw Gen.	Mvar Gen.	To Bus	Mw Flow	Mvar Flow	Mva Flow
1	1.000	0.000	100.00	20.00	89.67	60.71				
							2	-58.99	30.74	66.52
							5	48.67	9.97	49.68
2	1.000	3.434	100.00	20.00	196.88	-27.39				
							1	60.79	-29.15	67.42
							4	36.09	-18.24	40.44
3	1.000	-3.887	100.00	20.00	89.12	90.74				
							5	-17.35	42.84	46.22
							6	6.47	27.90	28.64
4	1.000	1.384	100.00	20.00	235.00	23.55				
							2	-35.45	17.52	39.54
							5	72.68	-0.65	72.68
							6	97.77	-13.33	98.67
5	0.972	-2.036	100.00	50.00	0.00	0.00				
							1	-47.67	-9.92	48.69
							3	18.24	-43.01	46.71
							4	-70.57	2.93	70.63
6	0.974	-3.512	100.00	10.00	0.00	0.00				
							3	-6.12	-29.15	29.78
							4	-93.88	19.15	95.82
Total	MW Gen	MVar Gen	MW Load	MVar Load	I2R Loss	I2X Loss	Charg MVar			
	610.7	147.6	600.0	140.0	10.7	21.3	13.7			

Table 3.3: SQP refinement of the GAOPF power cost optimization.

From Bus	Volt. Mag.	Volt. Angle	Mw Load	Mvar Load	Mw Gen.	Mvar Gen.	To Bus	Mw Flow	Mvar Flow	Mva Flow
1	1.000	0.000	100.00	20.00	91.23	59.99				
							2	-58.51	30.46	65.96
							5	49.74	9.53	50.64
2	1.000	3.405	100.00	20.00	196.29	-27.14				
							1	60.27	-28.93	66.86
							4	36.02	-18.21	40.36
3	1.000	-4.045	100.00	20.00	86.44	92.29				
							5	-18.94	43.76	47.68
							6	5.38	28.53	29.03
4	1.000	1.359	100.00	20.00	236.91	22.88				
							2	-35.38	17.49	39.47
							5	73.33	-0.87	73.33
							6	98.97	-13.74	99.92
5	0.972	-2.097	100.00	50.00	0.00	0.00				
							1	-48.70	-9.41	49.60
							3	19.88	-43.81	48.11
							4	-71.18	3.22	71.25
6	0.974	-3.603	100.00	10.00	0.00	0.00				
							3	-5.02	-29.76	30.18
							4	-94.98	19.76	97.02
Total	MW Gen	MVar Gen	MW Load	MVar Load	I2R Loss	I2X Loss	Charg MVar			
	610.9	148.0	600.0	140.0	10.9	21.8	13.7			

$$V_{i_{min}} \leq |V_i| \leq V_{i_{max}}, \qquad \forall\, Buses\ and\ W_l = 500.0$$

Where the variable W_l is the weight assigned to power losses and can be varied to minimize the power losses at the higher production expense. When the objective function is the sum of production cost and power losses, GAOPF arrived to an optimal point close to the one found when only the production cost was used. This is understandable since the power losses are normalized and as such, much smaller than the production cost part of the objective function. The SQP algorithm failed to converge using the same objective function as GAOPF. However, SQP was able to consistently refine the optimal point found by GAOPF when the GAOPF optimal point was used as the starting point for SQP. This example demonstrates that multiple goals cannot be simultaneously achieved. Obviously, a simple sum of individual objective functions is not the best approach. Tables 3.6 and 3.7 show the GAOPF and SQP solutions minimizing the production cost and power losses at the same time.

3.4.4 Minimizing Production Cost and Controlling Line Congestion

Under the deregulation rules, transmission must be available to every one on an equal basis. Often this can cause line congestions not because of a lack of transmission capacity in the system but because of the economic laws moving power from one particular part of the system to another. Computing Available Transfer Capability (ATC) is discussed elsewhere [1, 25]. Our goal is to control power transfer within a system with sufficient total transfer capability. In the following example, GAOPF is used to lower

Table 3.4: GAOPF power losses optimization.

From Bus	Volt. Mag.	Volt. Angle	Mw Load	Mvar Load	Mw Gen.	Mvar Gen.	To Bus	Mw Flow	Mvar Flow	Mva Flow
1	1.000	0.000	100.00	20.00	131.75	35.91				
							2	-0.77	-0.62	0.98
							5	32.52	16.53	36.48
2	1.000	0.044	100.00	20.00	100.80	17.60				
							1	0.77	-1.38	1.58
							4	0.03	-1.02	1.02
3	1.000	0.200	100.00	20.00	188.39	39.45				
							5	35.96	14.90	38.92
							6	52.43	4.55	52.63
4	1.000	0.042	100.00	20.00	182.91	40.98				
							2	-0.03	-0.98	0.99
							5	33.25	16.18	36.98
							6	49.69	5.79	50.02
5	0.973	-1.119	100.00	50.00	0.00	0.00				
							1	-31.97	-17.38	36.39
							3	-35.34	-15.61	38.63
							4	-32.69	-17.01	36.85
6	0.975	-2.134	100.00	10.00	0.00	0.00				
							3	-51.32	-4.28	51.49
							4	-48.68	-5.72	49.02
Total	MW Gen	MVar Gen	MW Load	MVar Load	I2R Loss	I2X Loss	Charg MVar			
	603.8	133.9	600.0	140.0	3.8	7.7	13.7			

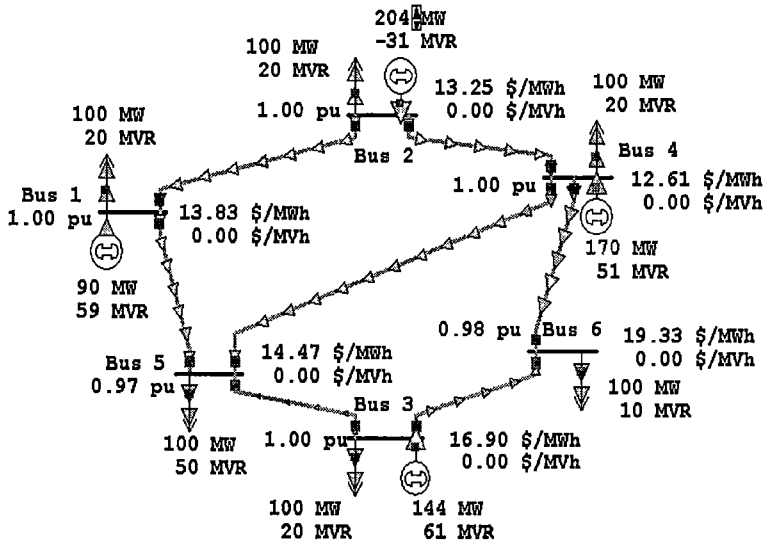


Figure 3.5: Power World 7.0 minimizing production cost while enforcing line flow limits.

the congestion on a line of our choice. As a matter of fact, we will not lower the line flow but keep it between the lower and upper limit. This type of constraint is possible using classical methods but is difficult to implement. In this example, the same network as in the previous examples is used. From the production cost minimization results in Table 3.3, we see that the highest power transfer is occurring the from bus number 4 to the bus number 6. The power transfer of 0.98 per unit will be set to be between 0.6 and 0.7 per unit. We use the same penalty function format that we used for voltage penalty but with different settings. The penalty function settings shown in Figure 2.2 are set for the nominal power transfers of 0.6 and 0.7, for the power change before maximum penalty is reached of 0.05, and the maximum penalty of 100.0. This means that the penalty is applied if the transfer exceeds 0.7 and reaches 100.0 at 0.8 transfer. Also, the penalty

Table 3.5: SQP refinement of GAOPF power losses optimization.

From Bus	Volt. Mag.	Volt. Angle	Mw Load	Mvar Load	Mw Gen.	Mvar Gen.	To Bus	Mw Flow	Mvar Flow	Mva Flow
1	1.000	0.000	100.00	20.00	133.91	34.87				
							2	-0.00	-1.00	1.00
							5	33.91	15.87	37.43
2	1.000	0.000	100.00	20.00	100.00	18.00				
							1	0.00	-1.00	1.00
							4	-0.00	-1.00	1.00
3	1.000	0.000	100.00	20.00	184.97	41.03				
							5	33.91	15.87	37.44
							6	51.06	5.16	51.32
4	1.000	0.000	100.00	20.00	184.97	40.03				
							2	0.00	-1.00	1.00
							5	33.91	15.87	37.44
							6	51.06	5.16	51.32
5	0.973	-1.200	100.00	50.00	0.00	0.00				
							1	-33.33	-16.67	37.27
							3	-33.33	-16.67	37.27
							4	-33.33	-16.67	37.27
6	0.975	-2.255	100.00	10.00	0.00	0.00				
							3	-50.00	-5.00	50.25
							4	-50.00	-5.00	50.25
Total	MW Gen	MVar Gen	MW Load	MVar Load	I2R Loss	I2X Loss	Charg MVar			
	603.8	133.9	600.0	140.0	3.8	7.7	13.7			

is applied if the transfer is lower than 0.6 and reaches 100.0 at 0.5. Note that only the upper or lower limit can be enforced by using one–sided penalty. Mathematically, the objective function and the constraints are defined as:

$$Cost = \sum_{i=1}^{all\ generators} \alpha_i + \beta_i * P_i + \gamma_i * P_i^2 \tag{3.4}$$

subject to:

$$V_{i_{min}} \leq |V_i| \leq V_{i_{max}}, \qquad \forall\, Buses\ and\ 0.6 \leq LineFlow_{46} \leq 0.7$$

From the results in Table 3.8 it is clear that line flow was shifted from the line connecting bus 4 and bus 6 to the rest of the network. The amount of this flow which is barely over the maximum limit at which the penalty starts indicates steep penalty function or equivalently, that this constraint has higher priority over other constraints. The production cost in this case is $7922.94 which is significantly higher than when only the production cost was minimized. However, power losses did not change much. Figure 3.5 shows Power World 7.0 solution with the same line flow constraint which resulted in production cost of $7941.53, again higher than the result obtained with the GAOPF.

3.5 Optimal Power Flow Summary

In the previous sections the GAOPF was demonstrated using different objective functions and compared with classical and SQP results. Power Flow was comparable only in the case where the power production cost was minimized and GAOPF performed slightly better than Power Flow. The next two examples use the same network to minimize in

Table 3.6: GAOPF power cost and losses optimization.

From Bus	Volt. Mag.	Volt. Angle	Mw Load	Mvar Load	Mw Gen.	Mvar Gen.	To Bus	Mw Flow	Mvar Flow	Mva Flow
1	1.000	0.000	100.00	20.00	92.66	58.63				
							2	-54.50	28.16	61.35
							5	47.17	10.46	48.31
2	1.000	3.168	100.00	20.00	190.03	-24.39				
							1	56.03	-27.11	62.24
							4	34.00	-17.29	38.14
3	1.000	-3.493	100.00	20.00	98.73	85.26				
							5	-12.21	39.89	41.71
							6	10.95	25.37	27.63
4	1.000	1.236	100.00	20.00	228.05	25.72				
							2	-33.43	16.43	37.25
							5	68.60	0.92	68.61
							6	92.88	-11.63	93.60
5	0.973	-1.953	100.00	50.00	0.00	0.00				
							1	-46.22	-10.52	47.41
							3	12.94	-40.37	42.40
							4	-66.72	0.90	66.72
6	0.975	-3.387	100.00	10.00	0.00	0.00				
							3	-10.62	-26.67	28.71
							4	-89.38	16.67	90.92
Total	MW Gen	MVar Gen	MW Load	MVar Load	I2R Loss	I2X Loss	Charg MVar			
	609.5	145.2	600.0	140.0	9.5	18.9	13.7			

Table 3.7: SQP refinement of GAOPF power cost and losses optimization.

From Bus	Volt. Mag.	Volt. Angle	Mw Load	Mvar Load	Mw Gen.	Mvar Gen.	To Bus	Mw Flow	Mvar Flow	Mva Flow
1	1.000	0.000	100.00	20.00	92.97	58.50				
							2	-54.14	27.95	60.93
							5	47.10	10.55	48.27
2	1.000	3.147	100.00	20.00	187.55	-23.27				
							1	55.64	-26.94	61.82
							4	31.91	-16.33	35.85
3	1.000	-3.574	100.00	20.00	95.85	86.89				
							5	-13.63	40.70	42.92
							6	9.48	26.19	27.86
4	1.000	1.332	100.00	20.00	233.27	23.43				
							2	-31.41	15.33	34.95
							5	70.21	0.28	70.21
							6	94.47	-12.19	95.25
5	0.972	-1.948	100.00	50.00	0.00	0.00				
							1	-46.16	-10.61	47.37
							3	14.40	-41.10	43.55
							4	-68.23	1.71	68.26
6	0.974	-3.380	100.00	10.00	0.00	0.00				
							3	-9.15	-27.48	28.96
							4	-90.85	17.48	92.52
Total	MW Gen	MVar Gen	MW Load	MVar Load	I2R Loss	I2X Loss	Charg MVar			
	609.6	145.5	600.0	140.0	9.6	19.3	13.7			

one case just the power losses and in the other case both power losses and power cost. In these two cases, the results could not be compared with Power Flow results since it does not provide this type of OPF. The generic Sequential Quadratic Programming algorithm had difficulties with convergence in all the cases but when GAOPF solution is used as a starting point, SQP is usually able to refine the GAOPF solution. Such SQP behavior is expected considering the mathematical algorithm used as well as genetic algorithm properties. Tables 3.9, 3.10, and 3.11 summarize the GAOPF results for the Power World example.

When comparing these three tables, it is clear that a compromise is made between minimizing power losses and power production cost. This is also an example of a multiobjective genetic algorithm optimization where the value of the objective function is a weighted sum of each objective. Scaling each objective can be used to give more weight to a particular goal or to just have all the goals contribute approximately equally to the final objective function value. A detailed mathematical analysis of power system markets [11] showed that the operating cost increases as the congestion increases. The last example minimizes the power production cost while controlling the congestion of the most congested line in the case of production cost minimization.

The line flow in this case is at the maximum no penalty limit. The conclusion is that the GAOPF does a good job minimizing the congestion but still keeps the line flow at the maximum limit allowing production cost to be optimized as much as possible while the line flow limit is obeyed. From all these examples, it seems that the GA can be

Table 3.8: GAOPF power cost minimization and line congestion control.

From Bus	Volt. Mag.	Volt. Angle	Mw Load	Mvar Load	Mw Gen.	Mvar Gen.	To Bus	Mw Flow	Mvar Flow	Mva Flow
1	1.000	0.000	100.00	20.00	86.96	60.37				
							2	-51.82	26.63	58.26
							5	38.78	13.74	41.14
2	1.000	3.010	100.00	20.00	194.15	-26.32				
							1	53.20	-25.87	59.15
							4	40.96	-20.45	45.78
3	1.000	-1.524	100.00	20.00	144.81	60.55				
							5	12.73	26.36	29.27
							6	32.08	14.20	35.08
4	1.000	0.686	100.00	20.00	180.88	45.26				
							2	-40.13	20.09	44.88
							5	50.61	8.29	51.28
							6	70.41	-3.12	70.48
5	0.973	-1.480	100.00	50.00	0.00	0.00				
							1	-38.09	-14.31	40.69
							3	-12.37	-27.58	30.22
							4	-49.55	-8.12	50.21
6	0.975	-2.675	100.00	10.00	0.00	0.00				
							3	-31.57	-15.14	35.02
							4	-68.43	5.14	68.62
Total	MW Gen	MVar Gen	MW Load	MVar Load	I2R Loss	I2X Loss	Charg MVar			
	606.8	139.9	600.0	140.0	6.8	13.6	13.7			

Table 3.9: OPF Summary with Power Cost Minimization.

	Production Cost	Power Loss
GAOPF	\$7824.57	10.668MW
SQP	\$7824.36	10.875MW
Power World	\$7868.40	16.5MW

Table 3.10: OPF Summary with Power Losses Minimization.

	Production Cost	Power Loss
GAOPF	$8198.7	3.843MW
SQP	$8187.63	3.837MW
Power World	N/A	N/A

Table 3.11: OPF Summary with Power Cost and Power Losses Minimization.

	Production Cost	Power Loss
GAOPF	$7828.7	9.473MW
SQP	$7827.29	9.644MW
Power World	N/A	N/A

successfully used to minimize (or maximize) a multiobjective function based on almost any system variables.

3.6 Genetic Algorithm Based Unit Commitment

In this section, a Genetic Algorithm is applied to the Unit commitment problem with simple objective functions and constraints. Figure 3.6 shows a hypothetical aggregated daily load for a power system for which we want to develop a unit commitment schedule. A basic genetic algorithm with binary string genes will be used to optimize the schedule using two different objective functions for demonstration purposes. The first objective function is very simple and its only goal is to find a combination of generating units every hour to produce enough power for the customers. The second objective function also takes into account minimum up–time. In this case the minimum up-time was taken to be

Table 3.12: Units available for scheduling.

Unit Number	Unit Capacity
Unit 1	100.0
Unit 2	300.0
Unit 3	500.0
Unit 4	600.0
Unit 5	700.0

the same for all the units and equal to two hours. Genetic Algorithm unit commitment is not restricted to these goals or with these constraints only as it is demostrated in [27]. Table 3.12 shows the capacities of the units used for the scheduling. Table 3.13 shows the results of the optimization process with the simple objective function. Table 3.14 shows the scheduling results using objective function with up–time constraints. Note that the only instances when the up–time is shorter than two hours is at the beginning and possibly at the end of the 24 hour scheduling period. If the scheduling is to run constantly, there would be no end of the 24 hour period and no violations of the minimum up–time scheduling.

3.7 Genetic Algorithm Drawbacks and Workarounds

A Genetic Algorithm is very computationally intensive and as such hard to use in on–line operations. In the case of an OPF, the main computational burden comes from the repeated calculation of the system's power flow. This implies usage of faster power flow methods to reduce the number of computations. Rather than going with an obvious

Table 3.13: Simple unit commitment over a 24hr period.

Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Load	Commitment
Off	On	Off	Off	Off	200	300
On	On	Off	Off	Off	300	400
Off	Off	On	Off	Off	400	500
On	Off	On	Off	Off	500	600
Off	Off	Off	Off	On	600	700
Off	On	On	Off	Off	700	800
Off	On	Off	On	Off	800	900
Off	On	Off	Off	On	900	1000
On	On	Off	Off	On	1000	1100
On	Off	On	Off	On	1200	1300
Off	On	On	Off	On	1400	1500
Off	On	Off	On	On	1500	1600
Off	On	On	On	Off	1350	1400
Off	On	Off	On	On	1500	1600
On	On	On	On	Off	1400	1500
Off	On	On	On	Off	1300	1400
Off	Off	Off	On	On	1200	1300
Off	Off	On	Off	On	1100	1200
Off	On	Off	Off	On	900	1000
On	On	On	Off	Off	800	900
On	Off	On	Off	Off	500	600
Off	Off	On	Off	Off	400	500
On	On	Off	Off	Off	300	400
Off	On	Off	Off	Off	200	300

Table 3.14: Unit commitment with minimum up–time over a 24hr period.

Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Load	Commitment
Off	On	Off	Off	Off	200	300
On	On	Off	Off	Off	300	400
Off	Off	On	Off	Off	400	500
On	Off	On	Off	Off	500	600
On	Off	Off	On	Off	600	700
Off	On	Off	On	Off	700	900
On	On	On	Off	Off	800	900
On	Off	On	On	Off	900	1200
Off	Off	On	On	Off	1000	1100
On	Off	On	Off	On	1200	1300
On	On	On	Off	On	1400	1600
Off	On	Off	On	On	1500	1600
Off	On	On	On	Off	1350	1400
On	On	On	Off	On	1500	1600
On	On	On	Off	On	1400	1600
Off	On	On	On	Off	1300	1400
Off	Off	Off	On	On	1200	1300
Off	Off	On	Off	On	1100	1200
Off	Off	On	On	Off	900	1100
Off	On	Off	On	Off	800	900
Off	On	Off	On	Off	500	900
Off	Off	On	Off	Off	400	500
Off	Off	On	Off	Off	300	500
Off	On	Off	Off	Off	200	300

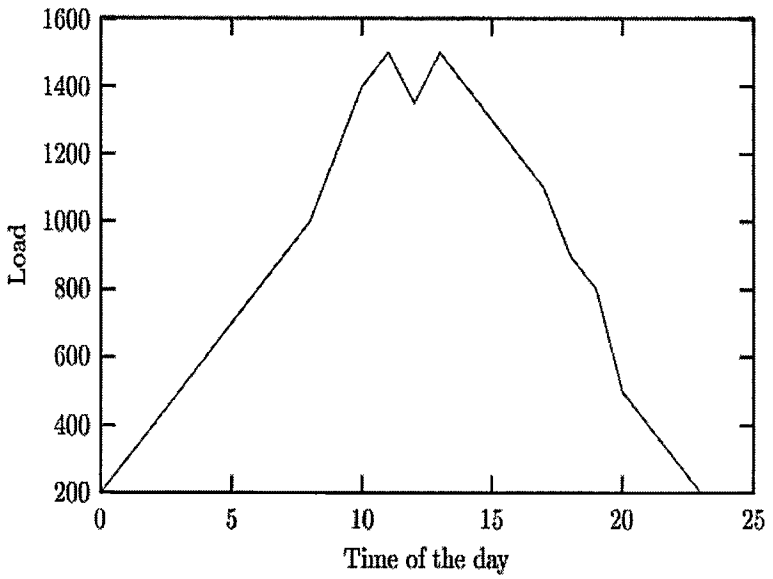


Figure 3.6: Daily load as a function of time.

solution that could quickly reach its limits and compromise accuracy, we will address this problem through GA parallelization and machine learning approaches.

A Genetic Algorithm is inherently a parallel algorithm. Solution populations can evolve in parallel, the fitness function for each individual can be calculated in parallel, or even power flow as the subproblem of the fitness function can be parallelized using standard linear algebra packages. Since the fitness function is the main computational difficulty, it is reasonable to distribute its computation in some way. Rather than relying on the availability of expensive parallel machines, it is assumed that a number of general purpose computers are available on a reasonably fast network. The underlying computational protocol is based on the Parallel Virtual Machine (PVM) [4] which includes message passing, and synchronization programming interface as well as architecture in-

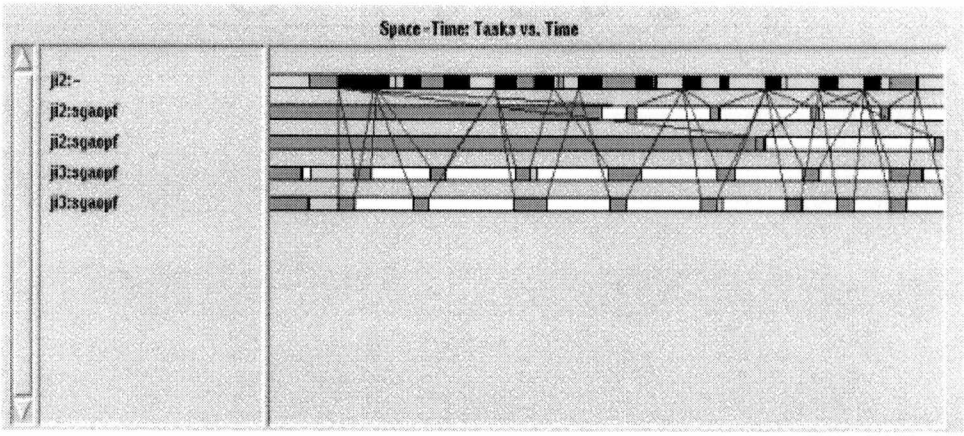


Figure 3.7: Distributed GAOPF space time dependence.

dependent communication of data. When a number of heterogeneous computers are exchanging data, it is important to take into account little and big endian and different floating point formats. Fortunately, PVM provides a unique representation of basic data types and a means of converting to/from a native computer format.

Distributing fitness function computations is relatively simple. Computation time scales linearly with the number of computers on the network. Figure 3.7 shows a space time–graph of a GAOPF running the fitness function distributed version of GAOPF in a master–slave mode. One computer is the master running statistical accounting, genetic operators, and distributing the individuals to available slave computers to calculate the fitness of each individual. Slave computers return the fitness value and the individual associated with it. Machine learning methods in power system operations will be discussed after decision tree introduction in the next chapter.

Chapter 4

Decision Tree Theory

Genetic programming can be computationally intensive and not usable in real time if the power system is very large and the objective function is complex. This is especially true in the UC case. However, the optimization can be done off–line and a decision tree can be trained to decide which units should be committed considering the predicted load over the next few hours. Note that load prediction and estimation is a completely different problem. In the unit commitment problem, it is assumed that the future load is known.

4.1 Decision Tree Inductive Learning

The original decision tree algorithm was proposed by Quinlan [26] in 1982 and since then applied to a number of artificial intelligence problems. The goal of the algorithm is to classify sample vectors of continuous and discrete type variables into disjoint classes.

This is achieved by learning from examples. A large set of examples belonging to a number of classes is provided as a training set. The training set is used to produce a series of "if-then-else" tests to classify instances that have training set characteristics. If the training set is large enough and if the samples to be classified are not significantly different than the training set examples, the procedure can correctly classify the samples which are not part of the training set. The type of the test ("if-then-else") defines a binary type tree. However, the decision trees are not limited to binary trees, but since any multi–branch tree can be reduced to a binary tree, it is reasonable to use the binary structure due to its simplicity.

There are different ways of processing the training set to produce a decision tree, but different strategies produce different trees which have almost the same classification characteristics but very different sizes. When the training set consists of thousands of examples, the size of the tree might be impractical. The common idea is to split the training set using "less than" in the case of continuous variables and "equal to" in the case of categorical tests until all the examples are classified correctly. It is impossible to prove that one method results in a better decision tree than another approach. The decision trees produced by the ID3 algorithm were compared to decision trees produced by other algorithms and found to be of considerably smaller size. The ID3 algorithm uses an approach from information theory to minimize the entropy after each splitting until the final subsets belong to a single class. At this point, the entropy is equal to zero.

The conditional entropy of classification, $E(A,q)$, is defined as:

$$E(A,q) = -\sum_{r\in\{yes,no\}} \frac{n_r}{n} \sum_{k\in C} \frac{n_{r,k}}{n_r} log_2 \frac{n_{r,k}}{n_r} \tag{4.1}$$

where n is the number of data objects in the sample set, C is the set of data classes, n_r is the number of objects with query result r and $n_{r,k}$ is the number of objects with query result r which are members of class k. It is clear that, at each node, the attribute A_i and threshold α_{ij} that yield the minimum conditional entropy provide the largest information gain for the query and therefore the best partitioning of the sample set.

From the description of the ID3 algorithm it should be noted that any classification function can be represented by an ID3 induced decision tree. The ID3 algorithm also prefers shorter trees over larger ones. The ID3 always uses only one hypothesis resulting in a locally rather than a globally optimal decision tree. ID3 has an inductive bias toward decision trees with fewer leaves [23] which might not be a desirable feature. After a short example in the next section, we will discuss how these ID3 features affect its performance.

4.2 Decision Tree Example

The following simple example is generated by the ID3 algorithm as defined by Quinlan. This example shows a training set and the decision tree derived from it. The vector $C[]$ denotes a continuous variable while the $D[]$ represents a discrete variable vector. A

quantitative test is applied to $C[]$ vector while a logical test is performed on the variables of the vector $D[]$. The input data follows the same format used by the machine learning database repository at The University of California, Irvine [24]. In this format, each database consists of two files, a file of variable names and description and a file of data samples. The file of names defines the variable names and their types. If a variable is of a discrete type, all possible values are listed. For continuous variables, a variable type is defined with the word "continuous". The data samples file contains the samples associated with the classes they belong to. The next example is defined by the following two files:

```
| This file was automatically generated from short1.ts

0, 1.

dis0:   0,1.
dis1:   0,1.
dis2:   0,1.
con0:   continuous.
con1:   continuous.
con2:   continuous.
```

and the data is given as:

```
1, 0, 0, 0.159204, -0.0763336, -0.0828701, 0.
1, 1, 1, -0.091528, 0.0481959, 0.0433321, 1.
1, 0, 0, 0.15842, -0.077602, -0.0808179, 0.
1, 0, 0, 0.156341, -0.0773795, -0.0789618, 0.
0, 1, 1, -0.342948, 0.171457, 0.171491, 1.
1, 0, 0, 0.157442, -0.0795252, -0.0779168, 0.
1, 0, 0, 0.157047, -0.0801064, -0.0769403, 0.
1, 1, 1, -0.0845121, 0.0399007, 0.0446114, 1.
0, 0, 0, -0.0923167, 0.0430449, 0.0492717, 0.
1, 1, 0, 0.0333967, 0.165913, -0.19931, 1.
1, 0, 1, 0.0330964, -0.210149, 0.177052, 0.
1, 0, 0, 0.149832, -0.0911175, -0.0587146, 0.
```

```
1, 1, 0, 0.0309253, 0.165163, -0.196088, 1.
0, 0, 1, -0.217939, -0.0865924, 0.304532, 0.
1, 1, 0, 0.0338461, 0.162341, -0.196187, 1.
1, 0, 0, 0.150639, -0.0748573, -0.0757818, 0.
1, 0, 1, 0.031739, -0.213549, 0.18181, 0.
0, 1, 0, -0.200208, 0.251931, -0.0517238, 1.
1, 0, 0, 0.150011, -0.0830738, -0.0669374, 0.
1, 1, 1, -0.091105, 0.0327538, 0.0583512, 1.
1, 0, 0, 0.15427, -0.0905875, -0.0636826, 0.
1, 0, 0, 0.156614, -0.0922853, -0.0643284, 0.
0, 1, 1, -0.342511, 0.156697, 0.185815, 1.
1, 0, 0, 0.148408, -0.0893247, -0.0590834, 0.
```

and the following program is the induced decision tree from the above data:

```
/*
        This file was produced by dt
        The current date and time is Fri Jul 23 18:39:46 1999
        The training file is short1.ts
*/

int dTree(double C[ ], int D[ ])
{
  if(D[1] == 0){
    return 0;
  }else{
    return 1;
  }
} /* End of dTree() function */
```

The same decision tree is also represented graphically in Figure 4.1. In this particular example, the decision tree is represented by a series of C statements. Another possibility is to represent the decision tree as a data table. A very attractive representation for testing and simulation purposes would be *Matlab*© form. The most useful representation would depend on the underlying application. There is a significant compromise in speed and size between the executable code and tabular representation. Regardless of the output format, this example shows that the ID3 algorithm correctly identifies the

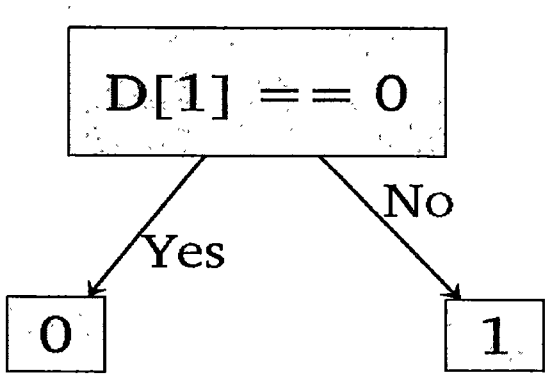


Figure 4.1: Decision tree induced from the training data.

most important decision variable $D[1]$ and produces the minimal tree possible.

It is important to emphasize the difference between the decision trees and other popular artificial intelligence methods such as neural networks and fuzzy logic. The decision trees do not suffer from false minima, exceedingly long training times or probability estimating problems. The decision tree approach is based on inductive learning while neural networks and fuzzy logic are based on optimization and searching techniques.

Decision trees can also be used for regression described as Classification and Regression Trees (CART) in [7]. However, there is a fundamental difference between the CART described in that report and the ID3 algorithm. First of all, ID3 can be used for classification only, while the CART implementation of decision trees can be used for regression as well as classification. Secondly, the CART method is based on probability and clustering theory. The ID3 method just partitions the sample space according to the provided examples. The ID3 classifies the training set with 100% accuracy while

the CART fails to perform with such accuracy even on the training set.

4.3 Decision Tree Issues

The decision trees are easy to induce from the training sets using an appropriate sample space partitioning criteria. Decision trees are used in many technical and nontechnical areas for data mining type applications. The requirements of each area has to be addressed by the decision tree capabilities. There are three important decision tree issues that need to be addressed when used in power systems. The first problem is ID3's over-fitting characteristic. The second issue is the accuracy of the decision tree classifier close to the decision surface. The third issue is how to handle large amounts of training data continuously generated on–line. These issues are identified and discussed but not dealt with in this work.

Over-fitting property is probably the most important issue with the ID3 algorithm. Due to its greedy optimization approach, ID3 is 100% accurate on the training set samples. This by no means implies that it is 100% accurate on a set of true testing samples from the same statistical distribution. In fact, ID3 performs poorly when compared with other algorithms derived from it. ID3 in its most basic form produces large trees overfitting the training data. A common way to deal with this problem is to perform post training pruning. The pruning process is basically a low pass filter smoothing out noisy data and spike samples. Usually, this process reduces the tree size and improves its accuracy.

The accuracy of the decision tree classifier close to the splitting surface depends on the statistical sample space distribution and splitting criteria. The decision trees, using an equality and threshold comparison, partition the entire sample space using stair–wise hyperplanes. For example:

```
if (C1 < 0.5)
  if (D1 == TRUE)
    return CLASS_A;
  else
    return CLASS_B;
else
```

This can be a problem for data points close to a hyperplane since they can be misclassified due to the non–smooth partitioning surface. This type of partitioning can also result in a large number of threshold comparisons in order to approximate a smooth surface. Sometimes, there is a linear or nonlinear relationship among the input variables implying that the partitioning could be much more effective and require less training samples if the tests were of the following form:

```
if (C1 < f(C1,C2,C3,...))
  if ((D2 && D1 && ...) == TRUE)
    return CLASS_A;
  else
   return CLASS_B;
else
```

Figure 4.2 illustrates an exact but nonlinear partitioning surface and its possible decision tree approximation. A possible partitioning surface in two dimensions based on the above reasoning would be a smooth curve. Revealing this type of relationship is more of an art than anything else. It is very likely that such dependences exist among power

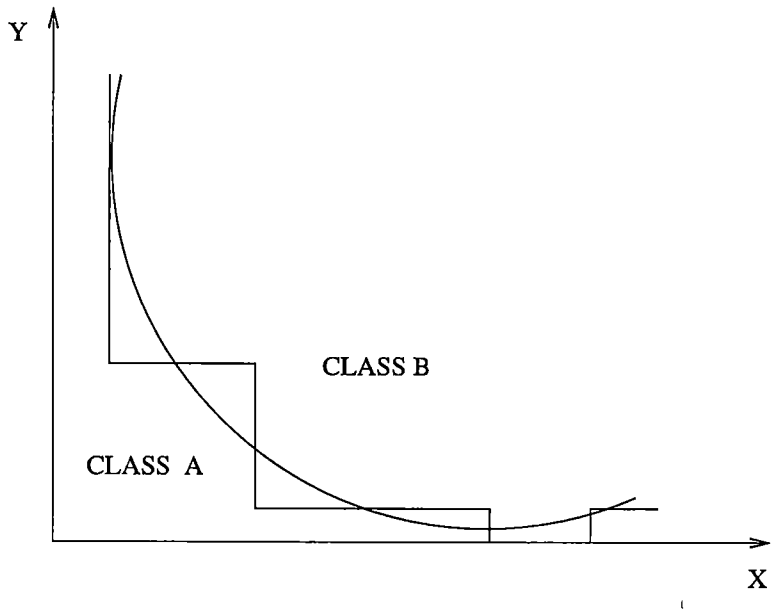


Figure 4.2: Decision Tree Classification Surface Approximation.

system variables but are not obvious and are difficult to incorporate in a decision tree induction algorithm. It is worth mentioning that CART suggests using this approach and that it is used in a simplified format in the $CART^{©}$ software package. These types of decision trees are known as oblique decision trees and a possible method for discovering planar decision surfaces is suggested in [28].

Usually, the decision trees are induced from a training set, cross–validated with the test sets, and then used to classify new samples. Occasionally, the training set is not fixed and might be increasing with time. In such cases, so called incremental decision trees could be used. The basic idea is to build a decision tree from the available training samples and incorporate new samples as they become available without rebuilding the tree from the beginning. This method is applicable to very large training sets and real

time operation since in such cases, the incremental learning time is much shorter than the time needed to rebuild the entire tree. A successful method of incremental induction of decision trees was reported in [31].

Here we pointed out three decision trees properties that could be exploited in power systems decision–making research. Due to their physical nature, power systems could probably benefit from the oblique decision trees. Incremental decision trees have an ability to continuously improve their classification accuracy with time. The state of a power system can be sampled periodically and the new samples incorporated in the existing tree. Both of these issues are state of the art research topics and left as a possible further improvement to decision tree use in power systems.

Chapter 5

GAOPF Based Unit Commitment and Machine Learning

Unit commitment (UC) is a difficult combinatorial optimization problem due to its high dimensionality and dependence on a nonlinear continuous optimization subproblem. As discussed in Appendix C, unit commitment must satisfy some basic requirements such as a unit schedule, which must be able to provide enough power for the load and system losses and meet additional operating constraints with respect to power quality and reliability. In addition to these requirements, some kind of optimality might be desirable. With the introduction of deregulation, generation cost minimization and profit maximization are becoming crucial issues. An ideal optimization algorithm would allow an arbitrary continuous subsolution to be used within the UC scheduling solution. The result is a set of generators supplying the load under optimal economic dispatch require-

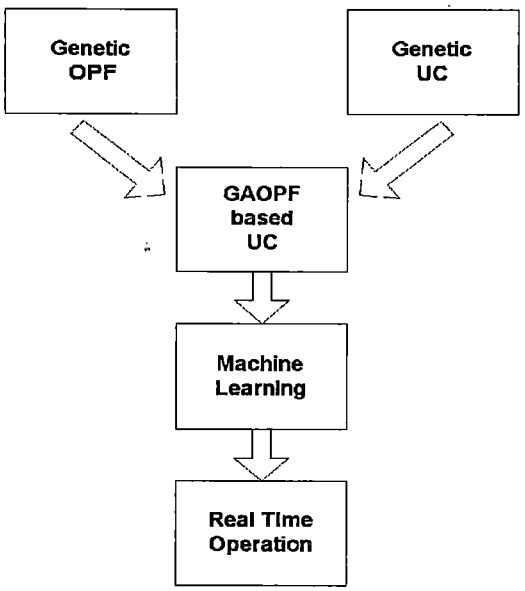


Figure 5.1: Genetic algorithm OPF based unit commitment.

ments. A number of different UC solutions are reviewed in the Appendix C, but none of them allow for a very flexible objective function used in the continuous subproblem. In this chapter, a genetic algorithm will be applied to both the combinatorial part and continuous optimization subproblem. The GAOPF algorithm discussed in Chapter 3 will be embedded in a binary string based genetic algorithm responsible for finding an optimal schedule. Since there are two optimization parts, a mixture of classical and AI methods can be used. However, this research uses AI algorithms in power systems so both parts will be addressed from an AI point of view. Since AI methods are computationally intensive, machine learning will be implemented allowing on–line use of the optimization results. As previously discussed, both UC and ED are viewed from different points of view in a deregulated market. As in the case of GAOPF based ED, we

will address UC from the point of view of an ISO. This means that the traditional unit constraints such as minimum up and down times are ignored. The following discussion can be easily adapted to an IPP point of view if so desired.

5.1 Genetic Algorithm Unit Commitment

In Chapter 3, genetic algorithms were applied to OPF as well as UC. The developed GAOPF can be used as a very flexible OPF for ED. The unit commitment example demonstrated the use of GA on a UC problem considering minimum up times and a minimum generation constraints. Now, a GAOPF and Genetic Algorithm UC (GAUC) will be merged to produce a flexible UC algorithm which can take into account different optimization goals and constraints. We will use the same network used in Chapter 3 to demonstrate GAOPF principles.

GAUC uses a binary string of length equal to the number of generators in the system minus one representing each generator being on or off. The swing bus is fixed by the system description file, reducing the length of the binary string by one. Therefore, the swing bus generator is a "must run" generator and it is also required that there is at least one more generator running. It is easy to add more "must run" or "preferred" generators effectively allowing the algorithm to decide what additional (possibly IPP) generators a utility should use. Since deregulation is still not regulated purely by market forces, a typical arrangement would be to have a number of "preferred" generators which are still regulated and supply the rest of the needed power from available but

more expensive IPPs. Whatever the number of "must run" and "preferred" generators used, GAUC searches for the optimal combination of additional generators needed. The optimality criteria is anything that can be optimized by GAOPF since the combinatorial part of GAUC has GAOPF embedded in its objective function. The usual goals would be minimizing the production cost, minimizing power losses, controlling particular line transmission, voltage levels, etc. We will use a simple goal of minimizing the production cost, or just power cost which is in the deregulated market a variable power/price curve depending on supply and demand. To demonstrate GAUC, the example network from Chapter 3 is used with bus and generator details given in the Tables 5.1 and 5.2. In this

Table 5.1: Network buses.

Bus Number	Bus Name	MW load	MVar Load	Low V	High V
1	BUS_1	0.5	0.2	0.95	1.05
2	BUS_2	0.5	0.2	0.95	1.05
4	BUS_4	0.5	0.2	0.95	1.05
3	BUS_3	0.5	0.2	0.95	1.05
5	BUS_5	0.5	0.5	0.95	1.05
6	BUS_6	0.5	0.1	0.95	1.05

example an arbitrary load was assigned to each bus. The top level genetic algorithm searches through the binary combinations for a combination of generators which result in a minimum generation cost while maintaining the bus voltages between the limits as described in the discussion of GAOPF operations. For the above data, the optimal combination is generator one (swing bus generator), generator two, and generator three. The swing generator and generator number four have the largest overall incremental cost

Table 5.2: Network generators.

No	P	V_{nom}	VAR_{max}	VAR_{min}	Cost	P_{min}	P_{max}
1	0.91	1.0	10.0	-10.0	$0.012 * x^2 + 12.0 * x + 105.0$	0	180
2	1.96	1.0	10.0	-10.0	$0.0096 * x^2 + 6.0 * x + 96.0$	0	180
3	0.86	1.0	10.0	-10.0	$0.013 * x^2 + 5.30 * x + 105.0$	0	180
4	2.37	1.0	10.0	-10.0	$0.0094 * x^2 + 9.4 * x + 94.0$	0	180

and should be expected to be excluded from a schedule under low loads. The swing generator is included since it is a "must run" generator and generator number four is excluded as expected. Since the GAOPF is a part of the GAUC, the optimal ED is a byproduct of this approach. This example is focused on an ISO point of view but it can easily be adjusted to include classical UC constraints. Table 5.3 shows a typical scheduling in addition to the swing bus generator. It is clear that the generator number four is on–line only after the total load exceeds some limit. From the previous discussion, this generator has the most expensive incremental cost and as such goes on–line last. For clearly illustrated reasons, the objective function is based only on the production cost and voltage limit penalties. Additional options available include minimization of power losses, and production cost and power losses and it is almost trivial to add other types of objective functions without any changes to the existing algorithm. The embedded genetic algorithm performs well when applied to the Unit Commitment problem. The only problem is that the GAUC is GAOPF embedded in another genetic algorithm resulting in a very computationally intensive program. Even for the small example above, running the GAUC a thousand times required almost ten hours. For systems of

Table 5.3: Unit Commitment as Seen by an ISO.

Unit 2	Unit 3	Unit 4	Load	Generation	Total Cost
On	On	Off	2.074	2.11338	1857.99
On	On	Off	2.84517	2.88223	2443.67
On	On	Off	2.89719	2.96767	2485.76
On	On	Off	3	3.05343	2558.39
On	On	Off	3.25377	3.34077	2826.97
On	On	Off	3.65716	3.71709	3214.4
On	On	Off	3.65677	3.72409	3272.82
On	On	On	4.04054	4.14141	3752.91
On	On	On	4.07843	4.12304	3801.4
On	On	On	4.20284	4.26573	3854.27

a few generators and a few buses, the delays might be acceptable but for large, realistic size networks, there must be a way to improve the performance. As in the GAOPF case, GAUC speed can be addressed using a parallel algorithm running on a network of non–dedicated computers or machine learning can be applied. Genetic Algorithm parallelization is a straight forward process as explained previously and will not be discussed further.

5.2 Machine Learning of the Unit Commitment

The two main ways of dealing with very high computational demands in this research are the use of parallel computations and machine learning. UC algorithm parallelization is, as described, straightforward. For the machine learning approach, the decision tree methodology is used. A training set of a thousand samples was simulated for the above example and the unit commitment schedule was produced using a GAUC. The load

at each bus was chosen from a uniform distribution between 0.1 and 1.1 per unit to simulate possible load peaking and to develop acceptable scheduling. Although the sample generation was very time consuming, decision tree induction was rather fast. The resulting decision tree is eleven levels deep and would take at most eleven real number comparisons to produce a scheduling decision. This is fast enough for real time operations. Since this is the ID3 algorithm, the accuracy is 100% as it should be when tested with the training set itself. There is no other supervised machine learning method with such accuracy even on the training set. When using a test case of approximately the same size as the training set and from the same uniform distribution, the misclassification rate was found to be 20.4617%. This is still a good accuracy compared with other machine learning methods. ID3 is not expected to perform with very high accuracy due to its overfitting features. Although, after a closer examination of the misclassified cases, it was obvious that almost all of the misclassified cases were very close to the partitioning surface. This indicates three important features of the UC problem. First, there are no "isolated points" in multidimensional space defined by the bus loads. The second implication is that there is a well defined surface partitioning the space in large regions. The third and final conclusion is that the accuracy of the decision tree can be improved by increasing the training set and refining the decision surface. Analysis of the physical problem showed that the generator controlled by this decision tree is the one with the highest production cost and as such would probably be the last one to be used. It became clear that the total load must be one of the most significant

decision variables. Therefore, the sum of all the loads was the decision curve which could improve the accuracy of the tree. After including the sum of all the loads in the training set, the misclassification dropped down to 3.6%. The critical misclassification variable in these 3.6% of all the cases was the sum of the loads in about 90% of the cases with very small critical distance. This behavior is exactly what should be expected from this example. Since a good partitioning surface is found, the misclassification rate should be much lower and all the errors involving this surface should be very small. Table 5.4

Table 5.4: Unit Commitment Schedule With Critical Loads.

Bus1	Bus2	Bus3	Bus4	Bus5	Bus6	On	Critical Load
0.46791	0.563381	0.147982	0.53963	0.971672	0.299345	0	B[4]=0.008307
0.495119	0.880418	0.59727	0.723838	0.956885	0.769663	1	B[3]=0.149101
0.129073	0.741853	0.731735	0.673739	0.937382	0.98781	1	B[3]=0.0146355
0.527415	0.673539	0.576653	0.214665	0.290406	0.26174	0	B[1]=0.113256
0.464469	0.731505	0.814713	0.284804	0.115542	0.320366	0	B[5]=0.0092375

shows a few examples consisting of bus loads, whether generator number four should be on or off and the critical load on a particular bus which could cause the on/off decision to change. If the load is changed above or below its present value by the critical load, the decision might or might not change. This is an interesting way of dealing with the load uncertainties. If the dispatcher decides that a particular load is approaching close to the decision surface, he/she can rerun the load scenario with the critical load changed by more than the critical value or it can be done automatically by the computer and a corrective action suggested. For example, in the second case, if load on the third bus

was changed by 0.15 per unit, it could change the status of the generator. However, a 0.15 per unit change in the bus load is a significant and probably unlikely change. On the contrary, in the fifth example, a change of only 0.01 could require the generator to be online. This critical value is within the load uncertainty levels and the dispatcher should investigate further. In this example, bus loads were used as the decision tree sample variables which are continuous quantities. There is nothing preventing the use of discrete variables representing status of a critical transmission line, for example. In that case, a $N-1$ based OPF could be used to train a decision tree and the critical variable would be whether the line is up or down.

In this section, three important features of the decision trees in UC are demonstrated. First is the speed of the execution of an induced decision tree. Second is the ability of the decision trees to estimate the most critical variable. These features make decision trees ideal for on–line operations and studies which require a very large number of runs. The third demonstrated feature is the existence of non–trivial decision surfaces and ability of the ID3 algorithm to take advantage of them when instructed to do so. Any problem solution which can identify these surfaces and achieve accuracy better than 95% with the ID3 algorithm indicates little need to search for a better machine learning solution. If the variable dependencies are not as obvious as in our example and sufficient accuracy cannot be achieved, ID3 is maybe not be the best decision tree algorithm to be used. It is used in this research as a proof of feasibility of the decision tree learning methodology. Decision trees using statistical distribution or clustering of the samples

as used in the $CART^{©}$ package might perform with significantly higher accuracy in some cases. Replacing ID3 with another algorithm in the GAOPF and GAUC should be straightforward.

5.3 Analysis of Genetic Algorithm Unit Commitment

As demonstrated in the previous section, a genetic algorithm can be successfully used for unit commitment in a complex setup. However, there are issues that must be addressed in a non–academic environment. First of all, it was noticed that a binary string GA has more difficulty converging requiring larger populations and a larger number of generations. This translates into a large increase of already significant computational times. A real variable GA has a better cost descending characteristic resulting in smaller populations and number of generations. It is possible to incorporate a GAOPF in a classical UC, such as dynamic programming based, or Lagrange relaxation based method. The benefit would be flexible optimization goals and a reasonably fast UC using straight forward modifications of the presented algorithm. In this research, different ways of improving computational times have been addressed. It was noted in Chapter 3 that there are two possible ways of increasing the speed of the genetic algorithm. One is using a large number of general purpose computers in parallel, and the second option is to perform off–line machine learning and exploit it in an on–line manner. The use of parallel computers for GAOPF computations was demonstrated in the same chapter. Exactly the same approach can be used for GAUC.

Supervised machine learning is learning from samples. A sample consists of a set of variables and a class to which that sample belongs. After learning is performed, classification is very fast, the unseen instances from the same distribution should be classified correctly, and critical decision limits are available for human examination.

Chapter 6

Conclusions and Future Research

Recent developments in the U.S. power industry indicated that there is no clear direction in which the deregulation should develop although the motivation and basic principles behind the deregulation are easy to understand. The main idea introduced by the government is to provide cheaper power through the free market competition. However, the Independent Power Producers (IPP) have different goals which are not cheap power but higher profit margins through power price regulated by the free economy laws of supply and demand. To promote free competition, the government requires utilities to unbundle their services into three independent entities: the generator owners, the transmission owners, and the Independent System Operators (ISO). The purpose of the unbundling is to prevent manipulating the market through the technical means. Recent events in California exposed some serious practical problems with the implementation of the deregulation [20]. Hopefully, a free and fair market will set the prices acceptable

by both the producers and consumers. A part of the U.S. power industry is presently deregulated, and it seems inevitable that the still regulated states will follow.

From the engineering point of view, the unbundling and the free market dictating the prices made some old problems more difficult and others easier to solve. Right now it is not clear what power system issues are the most important and how to handle them. Transmission congestion will definitely be a problem since the transmission owners and ISOs have limited control over bilateral contracts between the IPPs and consumers. How much transmission owners should charge for power wheeling without pancaked charges is another problem. Both problems are related to controlling power transmission within and between control areas. Flexible AC Transmission Systems (FACTS) [19] should make this problem easier to solve, but FACTS are expensive devices, still not in common use. On the other hand, the unbundling of the services separated some classical optimization tasks into IPP and ISO perspective making them simpler and easier to solve. For example, an IPP is not really concerned with the entire network stability. An IPP is concerned with running its generators in the most profitable way. However, an ISO's task is to maintain system reliability and possibly give price signals through the transmission control. ISOs should treat equally all IPPs but it can send price signals through transmission pricing or price–dependent load models as reported in [14]. An IPP must take into account such things as generator up/down times, maintenance schedules, and production cost curves. The ISO is more concerned with transmission congestion, operation reliability, and maintaining realistic prices rather than profit mar-

gins. Although, there are many unknowns regarding the power system operation under deregulation, in this dissertation some of the anticipated issues from the ISO perspective have been addressed. The ED and UC will have to be more flexible tools in the future allowing for different optimization goals. In the past, AI methods were able to solve many problems difficult to solve using classical methods. This work shows that the same is possible with the problems encountered in power system operations.

6.1 Conclusions

Two problems were investigated in this report. The first problem is Optimal Power Flow as seen by the ISO, and the second problem is Unit Commitment also as experienced by the ISO. Optimal Power Flow is a high dimensional, constrained, non–linear optimization problem. Usual solutions are based on linear optimization methods which involve linearization of the original problem in some way. As a result, a true optimal point might never be found or at least not the global optimal point. Here we used genetic algorithm based stochastic optimization to implement GAOPF and to compare it with the Newton–Raphson and LP OPF from Power World OPF software package. The comparable results obtained from GAOPF and Newton–Raphson were exactly the same. LP OPF produced considerably higher production cost. However, for some reason, the Newton–Raphson algorithm was discontinued in new versions of Power World software. GAOPF flexibility allowed us to explore different objective functions in addition to minimizing the production cost. Minimization of the power losses neglecting

the production cost produced considerably lower power losses but higher production cost, which is to be expected since minimizing the production cost is not equivalent to minimizing the power losses. Another cost function which is the sum of weighted production cost and power losses, resulted in a production cost between the minimal production cost and production cost found when only the power losses were minimized. Similarly, the power losses were between the minimal power losses found by minimizing the power losses only and the power losses when the production cost was minimized. To demonstrate the flexibility of the GAOPF approach, the production cost was minimized while the transmission of a congested line was set to be between prespecified limits. The production cost was higher than the minimal, but the transmission level was also between the desired levels. It is safe to say that GAOPF has good potential to be used in real life situations.

A genetic algorithm was also used for UC implementations. The use was demonstrated in two simple cases. In the first case, the only goal was to meet the production levels for the current hour. In the second case, minimum up-time was taken into account. No additional constraints as defined by the classical UC were used since they are largely ignored from the ISO perspective. The GAUC is done for a single time interval not taking into account past and present operation. Of course, this can be coupled with load forecasting to produce a long term UC schedule. In the deregulated environment, an ISO can take into account available transmission, demand, and supply to determine the unit commitment as well as other constraints imposed by the technology, available

resources, or regulators. Finally, to incorporate different constraints and optimization goals that could be raised by the deregulated environment, a GAOPF was embedded into the UC. This way, a UC can take into account many variables not addressed in the classical UC. The algorithm worked as expected producing a single time period unit commitment.

The GAOPF and GAUC perform well and allow for different constraints and goals to be explored. Unfortunately, the price to pay for such flexibility is long processing times. GAUC is particularly CPU time demanding because there is a genetic algorithm embedded into another genetic algorithm. There are two ways of dealing with this problem. The first possibility is to use more computers doing the calculations in parallel. The second option is to do supervised machine learning off–line and use the trained system in on–line operations. A genetic algorithm is inherently parallel algorithm and as such was paid little attention in this research. The supervised machine learning was explored using decision tree theory. A training set was produced off–line using GAUC and a decision tree was trained. When using a simulator to provide the training set, the size of the training set and the size of a test set is not an issue. Straight forward implementation of the ID3 algorithm resulted in a decision tree with 80% accuracy. After realizing that instead of a simple variable threshold a linear function of a combination of variables can be used, accuracy of almost 97% was achieved using the same training set. This is very high accuracy for any learning method.

6.2 Future Work

Genetic Algorithm and the Decision Tree learning produced reasonably good results to use this work for a further research. The main problem with the GAOPF and more so with GAUC is the long processing times. There are some obvious improvements and some not so obvious that can be investigated as noted below.

Both the GAOPF and GAUC use power flow to calculate the objective function value. Right now, the power flow algorithm used is the familiar Newton–Raphson method. The Newton–Raphson method results are the most accurate of all methods and it converges rather rapidly. The drawback is that it must calculate the Jacobian at each iteration and solve a system of non–linear equations. A decoupled power flow algorithm could dramatically decrease the CPU times at the expense of accuracy. How much accuracy is compromised is something that needs to be investigated on a case by case basis. A new solver can be substituted in GAOPF with minimal effort. Genetic Algorithm is easy to parallelize and distribute over a number of non–dedicated computers on the network. Parallelizing the existing algorithm should give good processing speed with the same computational results. In the case of GAUC, the decision tree algorithm can be used to provide real time performance. A major future research contribution could be done in this area. Most of the practically proven machine learning and optimization methods are reviewed in [21].

Decision tree induction is easy to understand, implement, and modify to fit different purposes. Decision trees can provide additional benefits not possible with other

learning methods. In chapter 5, we showed how the critical variable and its value can be estimated in the decision tree process. With minimal effort, explanatory decision making can be provided. These two features could be used during the real–life dispatch or for dispatcher training. For GAUC implementation, ID3 produced good results when the idea of so called oblique decision trees was incorporated. It seems that for physical systems using functions as splitting surfaces rather than single variable thresholds is very important. Unfortunately, discovering these linear or non–linear hyperplanes is largely an art and is highly problem dependent. Another possibility is using a different splitting criteria than that used in ID3. Splitting can be done using statistics principles as demonstrated in $CART^{©}$. However, considering the results in this work using ID3, it is questionable if an algorithm like $CART^{©}$ could achieve such high accuracy. $CART^{©}$ methodology could be used for load forecasting and other regression problems. Using it for an OPF would be a significant achievement but it needs further research. Power systems using decision trees could greatly benefit from incremental decision trees. The incremental decision trees are induced incrementally and are equivalent to the trees which would be induced non–incrementally but using the same training set. There are two benefits from using incremental decision trees. First, if the training set data is accumulated with time, it can reach a point where the tree induction from the beginning would take unacceptably long time. Adding a new sample to a tree should take much less time for large data sets. The second benefit is that if the data collected is increasing with time, the accuracy of the decision tree would increase.

Bibliography

Bibliography

[1] Available Transfer Capability Definitions and Determination. Technical report, North American Electric Reliability Council, June 1996.

[2] A. H. Mantawy, Youseff L. Abdel-Magid, and Shokri Z. Selim. Integrating Genetic Algorithm, Tabu Search, and Simulated Annealing for the Unit Commitment Problem. *IEEE Transactions on Power Systems*, 14(3):829–836, August 1999.

[3] Energy Information Administration. The Changing Stracture of the Electric Power Industry: Selected Issues. Technical report, 1998.

[4] Al Geist and Adam Beguelin. *PVM 3 User's Guide and Reference Manual*, 1994.

[5] Eric Allen and Marija Ilic. *Price–Based Commitment Decisions in the Electricity Market.* Springer, 1998.

[6] J. Arrilaga and C. P. Arnold. *Computer Analysis of Power Systems.* John Wiley & Sons, Inc., 1990.

[7] Charles J. Stone, Leo Breiman, Jerome H. Friedman, and Richard A. Olshen. *Classification and Regression Trees.* Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, California, 1984.

[8] Ching-Tzong Su and Wen-Tsuan Tyen. Enhanced Genetic Algorithms for Economic Dispatch Considering Special Unit Characteristics. *IEEE Power Tech'99 Conference, Budapest, Hungary*, 29 August 1999.

[9] IEEE Tutorial Course. Optimal Power Flow: Solution Techniques, Requirements and Challenges. 1996.

[10] Energy Information Administration. Financial Impacats of Nonutility Power Purchases on Investor-Owned Electric Utilities. Technical report, Energy Information Administration Office of Coal, Nuclear and Alternate Fuels, Washington, DC 20585, June 1994.

[11] Fernardo L. Alvarado. The Dynamics of Power System Markets. Technical Report PSERC-97-01, Department of Electrical and Computer Engineering, The University of Wisconsin, Madison, 12 March 1997.

[12] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Inc., 1988.

[13] I. S. Duff, R. G. Grimes, and J. G. Lewis. User's Guide for the Harwell–Boeing Sparse Matrix Collection (Release 1). Technical Report RAL-92-086. Technical report, Rutherford Appleton Laboratory, December 1992.

[14] James D. Weber and Thomas J. Overbye. Inclusion of Price Dependent Load Models in the Optimal Power Flow. In *Hawaii International Conference On System Sciences*, Kona, Hawaii, 6 January 1998.

[15] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. Super LU Users' Guide. Technical report, Computer Science Division, University of California, Berkeley, CA 94720, 21 September 1999.

[16] Nicolai M. Josuttis. *The C++ Standard Library: a Tutorial and Reference.* Addison-Wesley, July 1999.

[17] Jun Zhu and David L. Lubkeman. Object-Oriented Development of Software Systems for Power System Simulations. *IEEE Transactions on Power Systems*, 12(2):1002–1007, 2 May 1997.

[18] Kit Po Wong and Jason Yuryevich. Evolutionary-Programming-Based Algorithm for Environmentally-Constrained Economic Dispatch. *IEEE Transactions on Power Systems*, 13(2):301–306, May 1998.

[19] Marija Ilić and John Zaborski. *Dynamics and Control of Large Electric Power Systems.* John Wiley & Sons, Inc., 2000.

[20] Marija Ilic, Petter Skantze, and Poonsaeng Visudhiphan. Electricity Troubles in California: Who's Next? *IEEE Spectrum*, February 2001.

[21] Tom M. Mitchell. *Machine Learning.* McGraw–Hill, 1997.

[22] Mitsuo Gen and Runwei Cheng. *Genetic Algorithms & Engineering Optimization.* John Wiley & Sons, Inc., 2000.

[23] Neil C. Berkman and Tuomas W. Sandholm. What Should Be Minimized in a Decision Tree: A Re–Examination. Technical report, Computer Science Department, University of Massachusetts at Amherst, 6 September 1995.

[24] P. M. Murphy and D. W. Aha. *UCI Repository of Machine Learning Databases [Machine Readable Data Repository].* University of California, Department of Information and Computer Science, Irvine, CA, 1992.

[25] Sauer W. Peter. *Technical Challanges of Computing Available Transfer Capability (ATC) in Electrical Power Systems.* 1997.

[26] J. R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–106, 1987.

[27] S. A. Kazarlis, A. G. Bakirtzis, and V. Petridis. A Genetic Algorithm Solution to the Unit Commitment Problem. *IEEE Transactions on Power Systems*, 11(1), February 1996.

[28] Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A System for Induction of Oblique Decision Trees. *Journal of Artificial Intelligence Research*, August 1994.

[29] Tetsuzo Tanino, Yoshikazu Sawaragi, and Hirotaka Nakayama. *Theory of Multiobjective Optimization.* Academic Press, Inc., 1985.

[30] Robert J. Thomas and Thomas R. Schneider. *Underlying Technical Issues in Electricity Deregulation*, pages 6–9. 1997.

[31] Paul E. Utgoff. An Improved Algorithm for Incremental Induction of Decision Trees. Technical report, Department of Computer Science, University of Massachusetts, 1994.

[32] Visual Numerics, Inc. *IMSL C/Math Library 4.0 User's Guide*, 2000.

[33] Matthew Wall. *GAlib: A C++ Library of Genetic Algorithm Components.* Mechanical Engineering Department, Massachusetts Institute of Technology, 1996.

[34] Kevin Warwick. *Artificial Intelligence Techniques in Power Systems.* The Institution of Electrical Engineers, London, 1997.

[35] Daniel James Weber. Implementation of Newton–Based Optimal Power Flow into a Power System Simulation Environment, 1997.

[36] Jr. William D. Stevenson. *Elements of Power System Analysis.* McGraw-Hill Book Company, 1982.

[37] Allen J. Wood and Bruce F. Wollenberg. *Power Generation Operation and Control.* John Wiley & Sons, Inc., 1996.

[38] Xiaodong YIN. Application of Genetic Algorithms to Multiple Load Flow Solution Problem in Electrical Power Systems. In *Proceedings of the 32nd Conference on Decision and Control, San Antonio, Texas*, December 1993.

Appendices

Appendix A

Power Flow Review

Power flow is a steady–state analytical tool of a power system. The power flow within a system is completely known if real or active power (P_i), reactive power (Q_i), voltage magnitude (V_i), and voltage phase angle (θ_i) are known at each bus i. Only two variables are known at each bus and the goal is to solve for the other two variables. The three most common bus types are load bus (P, Q known), voltage controlled bus (P, $|E|$ known), and generator bus (P, $|E|$ known, or P, Q if Q exceeds generator limits). A power system is connected with lines of known complex impedance between the buses and the ground. The system can be described with a set of equations similar to any electrical circuit. However, since the known variables are complex power quantities, the power system equations are nonlinear.

The following equations are the derivation of Newton–Raphson solution [37] used in this work. The Newton–Raphson solution is based on the first order, multivariable

Taylor series expansion which requires one time calculation of the system's impedance matrix and an iterative calculation of the Jacobian. If the power system's nonlinear equations are represented by:

$$F(V) = const \tag{A.1}$$

we choose a starting value for the independent variable V^0. The starting value will cause an error, ε in function F:

$$F\left(V^0\right) + \varepsilon = const \tag{A.2}$$

The error can be decreased using Taylor series expansion,

$$F\left(V^0\right) + \frac{dF\left(V^0\right)}{dV}\Delta V + \varepsilon = const \tag{A.3}$$

solving for independent variable correction ΔV by setting error ε to zero and reiterating with the new value of the independent variable until the error is acceptable:

$$\Delta V = \left(\frac{dF\left(V^0\right)}{dV}\right)^{-1}\left[const - F\left(V^0\right)\right] \tag{A.4}$$

In power systems case, the independent variable is the set of complex value bus voltages, the derivative in the Taylor series expansion is the Jacobian of the power injection equations at each bus, and the error is the set of P_i and Q_i mismatches. A power

system's currents depend on the bus voltages in a linear manner:

$$\begin{bmatrix} I_1 \\ I_2 \\ . \\ . \\ I_n \end{bmatrix} = \begin{bmatrix} Y_{11} & Y_{12} & Y_{13} & Y_{14} & Y_{15} \\ Y_{21} & Y_{22} & Y_{23} & Y_{24} & Y_{25} \\ . & . & . & . & . \\ . & . & . & . & . \\ Y_{n1} & Y_{n2} & Y_{n3} & Y_{n4} & Y_{n5} \end{bmatrix} \begin{bmatrix} E_1 \\ E_2 \\ . \\ . \\ E_n \end{bmatrix} \tag{A.5}$$

where:

$$Y_{ij} = -y_{ij} \quad \text{and} \quad Y_{ii} = \sum_j y_{ij} + y_{ig}$$

over all i, j connected buses and i, g ground connections.

At each bus, the injected complex power is given by:

$$P_i + jQ_i = E_i I_i^* \tag{A.6}$$

where:

$$I_i = \sum_{k=1}^{N} Y_{ik} E_i^*$$

then

$$P_i + jQ_i = E_i \left(\sum_{k=1}^{N} Y_{ik} E_k \right)^*$$

and

$$P_i + jQ_i = |E_i|^2 Y_{ii}^* + \sum_{\substack{k=1 \\ k \neq i}}^{N} Y_{ik}^* E_i E_k^*$$

The last equation can be expanded into:

$$\begin{aligned} P_i + jQ_i &= \sum_{k=1}^{N} |E_i||E_k| \left(G_{ik} - jB_{ik}\right) e^{j(\theta_i - \theta_k)} \\ &= \sum_{k=1}^{N} |E_i||E_k| \left[G_{ik}\cos\left(\theta_i - \theta_k\right) + B_{ik}\sin\left(\theta_i - \theta_k\right)\right] \\ &\quad + j\sum_{k=1}^{N} |E_i||E_k| \left[G_{ik}\sin\left(\theta_i - \theta_k\right) - B_{ik}\cos\left(\theta_i - \theta_k\right)\right] \end{aligned} \tag{A.7}$$

where

$$\begin{aligned} \theta_i,\, \theta_k &= \text{the phase angles at buses } i \text{ and } k \\ |E_i|,\, |E_k| &= \text{the bus voltage magnitudes} \\ \mathrm{G_{ij}} + jB_{ik} &= Y_{ik} \text{ is the } ik \text{ term in the } Y \text{ matrix} \end{aligned}$$

The system of equations A.7 is a nonlinear system which is solved using standard first order Taylor series approximation. The Jacobian of the Equations in A.7 is given by the following matrix equation:

$$\begin{bmatrix} \Delta P_1 \\ \Delta Q_1 \\ \Delta P_2 \\ \Delta Q_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} \frac{\partial P_1}{\partial \theta_1} & \frac{\partial P_1}{\partial |E_1|} & \frac{\partial P_1}{\partial \theta_2} & \frac{\partial P_1}{\partial |E_2|} & \cdots \\ \frac{\partial Q_1}{\partial \theta_1} & \frac{\partial Q_1}{\partial |E_1|} & \frac{\partial Q_1}{\partial \theta_2} & \frac{\partial Q_1}{\partial |E_2|} & \cdots \\ \frac{\partial P_2}{\partial \theta_1} & \frac{\partial P_2}{\partial |E_1|} & \frac{\partial P_2}{\partial \theta_2} & \frac{\partial P_2}{\partial |E_2|} & \cdots \\ \frac{\partial Q_2}{\partial \theta_1} & \frac{\partial Q_2}{\partial |E_1|} & \frac{\partial Q_2}{\partial \theta_2} & \frac{\partial Q_2}{\partial |E_2|} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \Delta\theta_1 \\ \Delta|E_1| \\ \Delta\theta_2 \\ \Delta|E_2| \\ \vdots \end{bmatrix} \tag{A.8}$$

To further simplify the derivatives, instead of $\Delta|E_i|$, we use $\frac{\Delta|E_i|}{|E_i|}$ which results in:

$$\begin{aligned}
\frac{\partial P_i}{\partial \theta_k} &= |E_i||E_k|\left[G_{ik}\sin\left(\theta_i-\theta_k\right)-B_{ik}\left(\theta_i-\theta_k\right)\right] \\
\frac{\partial P_i}{\left(\frac{\partial |E_k|}{|E_k|}\right)} &= |E_i||E_k|\left[G_{ik}\cos\left(\theta_i-\theta_k\right)+B_{ik}\sin\left(\theta_i-\theta_k\right)\right] \\
\frac{\partial Q_i}{\partial \theta_k} &= -|E_i||E_k|\left[G_{ik}\cos\left(\theta_i-\theta_k\right)+B_{ik}\sin\left(\theta_i-\theta_k\right)\right] \\
\frac{\partial Q_i}{\left(\frac{\partial |E_k|}{|E_k|}\right)} &= |E_i||E_k|\left[G_{ik}\sin\left(\theta_i-\theta_k\right)-B_{ik}\cos\left(\theta_i-\theta_k\right)\right]
\end{aligned} \tag{A.9}$$

For $i = k$:

$$\begin{aligned}
\frac{\partial P_i}{\partial \theta_i} &= -Q_i - B_{ii}E_i^2 \\
\frac{\partial P_i}{\left(\frac{\partial |E_i|}{|E_i|}\right)} &= P_i + G_{ii}E_i^2 \\
\frac{\partial Q_i}{\partial \theta_i} &= P_i - G_{ii}E_i^2 \\
\frac{\partial Q_i}{\left(\frac{\partial |E_i|}{|E_i|}\right)} &= Q_i - B_{ii}E_i^2
\end{aligned}$$

And in the matrix form:

$$\begin{bmatrix} \Delta P_1 \\ \Delta Q_1 \\ \Delta P_2 \\ \Delta Q_2 \\ \vdots \end{bmatrix} = [J] \begin{bmatrix} \Delta\theta_1 \\ \frac{\Delta|E_1|}{|E_1|} \\ \Delta\theta_2 \\ \frac{\Delta|E_2|}{|E_2|} \\ \vdots \end{bmatrix} \tag{A.10}$$

Equation A.10 is in the format used by the Newton–Raphson iterative solver. The whole process starts by assigning an initial solution for complex voltages which is usually of a magnitude of 1.0 per unit and 0 phase. The next step is to calculate ΔP_i and ΔQ_i

as well as the Jacobian. Using these values, $\Delta\theta_i$ and $\frac{\Delta|E_2|}{|E_2|}$ are calculated and a new solution for voltages is found. This is repeated until change in P and Q is sufficiently small.

Appendix B

Optimal Power Flow Review

The Optimal Power Flow (**OPF**) problem has been a topic of intense research for the last forty years. OPF is a static power flow problem which tries to determine the optimal control variables in a power network under constraint conditions. The OPF can be mathematically formulated in general terms as in [9]:

$$Min \underset{z}{f}(Z), \quad Z \in \Re^{m+n}, \quad f : \Re^{m+n} \rightarrow \Re^{1} \tag{B.1}$$

subject to:

$$G(Z) = 0, \quad G : \Re^{m+n} \rightarrow \Re^{a} \tag{B.2}$$

$$H(Z) \leq 0, \quad H : \Re^{m+n} \rightarrow \Re^{b} \tag{B.3}$$

where $f(\mathbf{Z})$ is the objective function to be minimized, $G(\mathbf{Z})$ are the nonlinear power flow equations and $H(\mathbf{Z})$ are the nonlinear inequality constraints. Vector $\mathbf{Z} = [\mathbf{U}\,\mathbf{X}]^{\mathbf{T}}$

is the vector of both state and control variables. State variables vector **X** includes bus voltages, reference bus angle, non–controlled generator MVA and MVAR outputs and loads, fixed bus voltages, etc. Similarly, **U** is the vector of control variables such as real and reactive power generation, phase–shifter angles, net interchange, load shedding, DC transmission line flows, control voltage settings, LTC settings, etc. Equality and inequality constraints include limits on all control variables, power flow equations, generation/load balance, power flow limits, bus voltage limits, reserve limits, MVAR limits, etc. The optimization objective could include power cost optimization, loss minimization, minimum control–shift, minimum number of controls rescheduled, etc. With the introduction of the free market, a different set of constraints and different objectives are being introduced depending on the deregulation model.

OPF solutions are based on both linear and nonlinear numerical methods. Linear programming methods are based on different variants of Linear Programming (LP) while the nonlinear methods include Sequential Quadratic Programming, Augmented Lagrangian Methods, Generalized Reduced Gradient, Projected Augmented Lagrangian, Successive Linear Programming, Interior Point Methods, among others. The number of the different solutions and variants is a clear indication of the complexity of the problem. We will review the Sequential Quadratic Programming (SQP) method which is a modified Newton–Raphson method. All of the OPF algorithms in use in some way simplify the original problem.

B.1 Sequential Quadratic Programming Optimal Power Flow

Nonlinear optimization is based on various gradient approaches. The best known is the steepest descent method which requires determining the descent direction by solving a multidimensional equation based on the gradient matrix of the objective function. Another approach which resulted in many variations is based on the Newton–Raphson method. In the multidimensional minimization problems, we are looking for a root of $\nabla f(Z) = 0$. The Taylor series expansion applied to to finding a root of $\nabla f(Z)$ gives:

$$0 = \nabla f\left(Z^{k+1}\right) \approx \nabla f\left(Z^{k}\right) D^{k} + \nabla^{2} f\left(Z^{k}\right) D^{k} \tag{B.4}$$

which finds the descent direction ($D^k = Z^{k+1} - Z^k$) from the following equation:

$$\nabla^{2} f\left(Z^{k}\right) D^{k} = -\nabla f\left(Z^{k}\right) \tag{B.5}$$

where $\nabla^2 f\left(Z^k\right)$ is the Hessian matrix evaluated at Z^k. Calculating the Hessian is difficult and error prone task and the Hessian might not be positive–definite as required by the Newton method. The quasi-Newton method calculates an approximation of the Hessian from Z^k and gradient information from previous iterations resulting in the solution for D^k:

$$B^{k} D^{k} = -\nabla f\left(Z^{k}\right) \tag{B.6}$$

where B^k is the Hessian approximation matrix. In addition to descent direction, a step length α^k in the given direction must be found as well. The usual method of finding α^k is from the one dimensional minimization:

$$\underset{\alpha}{Min}\, f\left(Z^k + \alpha D^k\right) \tag{B.7}$$

To handle the constraints from equation B.1, a Lagrangian must be used:

$$L\left(Z, \lambda\right) = f\left(Z\right) - \sum_{i=1}^{a} \lambda_i g_i + \sum_{j=1}^{b} \lambda_{j+a} h_j \tag{B.8}$$

where $\lambda \in \Re^{a+b}$ is the vector of Lagrangian multipliers and h_i and g_i are elements of $H\left(Z\right)$ and $G\left(Z\right)$.

Sequential Quadratic Programing extends the quasi–Newton method by approximating the objective function and non–linear constraints using Taylor's series expansion and solving the resulting Quadratic Programming Problem. SQP approximates the objective function with a positive definite quadratic approximation:

$$q^k\left(D\right) = \nabla f\left(Z^k\right) D + \frac{1}{2} D^T \nabla^2_{ZZ} L\left(Z^k, \lambda^k\right) D \tag{B.9}$$

and D^k is calculated by solving the quadratic programming subproblem:

$$\underset{D}{Min}\, q^k\left(D\right) \tag{B.10}$$

subject to:

$$G\left(Z^k\right) + J\left(Z^k\right) D = 0 \tag{B.11}$$

$$H\left(Z^k\right) + I\left(Z^k\right) D \leq 0 \tag{B.12}$$

where J and I are Jacobians of the constraint vectors G and H. The next step Z^{k+1} is updated as $Z^{k+1} = Z^k + \alpha^k D^k$ where α^k is the step length that needs to be determined. Unlike in the unconstrained non–linear optimization, determination of the step length is a rather difficult task since in addition to choosing an optimal step it must also minimize constraint violations. The so called "merit function" is used to express the relative importance of these two goals. A good choice for a merit function is described in [9] but it is by no means the best one:

$$P\left(Z, \nu\right) = f\left(Z\right) + \sum_{i=1}^{a} \nu_i \left|g_i\right| + \sum_{j=1}^{b} \nu_{a+j} \max\left[h_j, 0\right] \tag{B.13}$$

where $\nu \in \Re^{a+b}$ is the vector of positive penalty parameters and g_i, h_j are elements of the constraint vectors $G\left(Z\right)$ and $H\left(Z\right)$ respectively. Additionally, $\nu_i \geq \left|\lambda_i\right|$, $i = 1, 2, ..., a, ..., a + b$ and finally, the step length is chosen to minimize $P_1\left(Z^k + \alpha D^k, \nu\right)$.

Appendix C

Unit Commitment

Unit commitment is a task in power systems operations where a decision has to be made regarding which power units need to be online at any point in time to be able to supply the current power demand. In the monopolistic market, it is sufficient to ensure that the load demand is covered. All the production and transmission cost is eventually recovered by the utility regardless of the production efficiency. In the new, deregulated market, this is not necessarily true. If a power producer fails to be price competitive it will sooner or later go out of business. Therefore, it is important that the unit commitment incorporates both the technical and economical constraints.

Technical constraints are the constraints that need to be obeyed to be able to produce enough power while providing good voltage and frequency stability, system integrity, and taking in account units' minimum up and down times. On top of these constraints, each power system, power pool, the reliability council and so forth may impose a set of

additional rules on scheduling of the units.

The main economic goal is profit maximization. Profit maximization is related to production cost minimization, but they are not equivalent. Minimizing the production cost is necessary but not sufficient to maximize the profits.

C.1 Classical Methods

There are a few popular approaches to the unit commitment problem. However, most of them are applicable to power networks with simplified power generation and loss models[37]. Usually, the model used is an infinite bus with the generators and a load connected to it. The goal is to find an optimal unit commitment schedule for a number of hours in advance that will minimize the generation cost. Unit commitment is not intended to minimize the losses taking into account the topology of the power system. As a consequence of this, it cannot cope with transmission line overloading, power wheeling and similar problems. However, even this simplified model should take into account the economic dispatch and include it in its objective function.

To illustrate the combinatorial complexity of the unit commitment problem, consider a power system with **N** units [37]. The following equations show the number of unit combinations that must be examined in a brute force approach to find the best unit combination for a particular load:

$$C(N,1) + C(N,2) + C(N,3) + ... + C(N,N-1) + C(N,N) = 2^N - 1 \qquad \text{(C.1)}$$

Table C.1: Number of possible Unit Commitment combinations over 24 hours period.

Number of Units N	*Number of Possible Combinations*
5	6.2×10^{35}
10	1.73×10^{72}
20	3.12×10^{144}
40	Too large

where $C(N, j)$ is the number of N items taken j at a time:

$$C(N,j) = \frac{N!}{(N-j)!j!}, \quad j! = 1 \times 2 \times ... \times j \tag{C.2}$$

For a period of M intervals, the number of combinations is:

$$(2^N - 1)^M \tag{C.3}$$

Table C.1 shows a number of possible unit commitment combinations over a period of 24 hours.

At this point, it should be clear that an exhaustive search is not a solution for a unit commitment problem of any realistic size. Even if a very simplistic cost function is used, just enumerating all the combinations is too demanding for today's computers. The following sections will introduce existing optimization methods as applied to the unit commitment problem.

C.1.1 Priority-List Method

The priority list method is the simplest unit commitment solution and as such can address only the most basic scenarios. The priority list method can solve simple problems where it is needed to determine the unit commitment based on the current measured or estimated load and on the predicted future load. The main idea is to have a priority list of power units and when the load changes, to activate or deactivate the next units on the list until the power demand is met. If the future load can be predicted, it can be incorporated in the algorithm by making simple decisions whether it is economically justifiable to change a unit status due to start up and shut down costs. This is about the only situation that the priority list approach can handle. It is very simple and needs very low computational resources.

The priority list approach can be described as follows assuming that the unit commitment is in a correct state [3]:

- At each period determine if dropping the next unit on the list would satisfy the load demand and other reserves. If not, continue with the present commitment. Otherwise go to the next step.
- If the unit to be dropped will be needed again in less than its minimum down time, continue with the present commitment. If not, go to the next step.
- If shutting down the unit and restarting it when needed later on is less costly then keeping it on line, shut it down. If not, keep it on line.

- Repeat the procedure for the next unit on the list.

C.1.2 Dynamic Programming Solution

Dynamic programming was introduced in the late 1950's to address various (**NP**) complete (non–polynomial complex) optimization problems. Its theoretical background is based on the calculus of variations and is rather complex. Software implementation of a dynamic programming algorithm is relatively simple once the problem is defined in terms suitable for dynamic programming. The basic idea behind the dynamic programming is to solve an N–dimensional problem as N one–dimensional problems. This is done in stages with the notion that the optimal solution at stage N must contain the optimal solution at stage $N-1$. In a sense, there are two boundary conditions, and the solution can start from either one. These approaches can be described as backward and forward solution. The unit commitment problem can be formulated in a way to be able to solve it using dynamic programming, and it makes more sense to do it in a forward manner since the system progresses forward in time. Usual assumptions made in dynamic programming solutions are as follows [37]:

- A state consists of an array of units with specified units operating and the rest off–line.
- The start-up cost of a unit is independent of the time it has been off-line.
- There are no costs for shutting down the units.

- There is a strict priority order, and in each interval a specified minimum amount of capacity must be operating.

There are at least two issues which make dynamic programming of limited use for most power system optimization problems. First of all, the cost function used in the dynamic programming must be separable in terms of optimizing variables. This immediately rules out use of power flow as a part of the cost function where each generation unit is a variable whose optimal production level needs to be found. Secondly, the cost function has a number of constraints. Each constraint increases the state dimension of the dynamic programming problem. A similar situation is if there are multiple decision variables at each stage.

Dynamic programming also suffers from "the curse of dimensionality" since at each stage, all possible unit combinations are examined. To reduce the computational complexity, dynamic programming can be implemented using priority lists. For example, a system with four units would have sixteen combinations at each stage. Using priority lists, this number is reduced to only four combinations per stage. Another simplification would be to consider only a number of combinations in a previous and current stage. Of course, this could result in a suboptimal unit commitment policy.

A typical unit commitment cost function to be minimized by the dynamic programming could look like the following one reported in [37]:

$$F_{cost} = \min_{\{L\}}[P_{cost}(K, I) + S_{cost}(K-1, L : K, I) + F_{cost}(K-1, L)]$$

where I and L are possible states at the time period K and

$F_{cost}(K, I)$ = least total cost to arrive to state (K, I)

$P_{cost}(K, I)$ = production cost for state (K, I)

$S_{cost}(K-1, L : K, I)$ = transition cost from state $(K-1, L)$ to state (K, I)

The transition cost function used in dynamic programming can be very flexible. The cost function used in [5], includes non–utility generation price in a stochastic environment. The next section introduces Lagrange Relaxation method which uses dynamic programming of a trivial size and consequently does not have the same problems as the dynamic programming but introduces new problems of its own.

C.1.3 The Lagrange Relaxation Solution

The dynamic programming method has a problem with large systems due to its enumeration approach at each stage. Frequently, to make the calculations possible in real time, the number of states searched is reduced according to some criteria. This means that the probability of finding the global optimal operating point is reduced.

The Lagrange relaxation technique does not suffer from this problem, but there are new problems typical for this method. The Lagrange relaxation method is based on a dual optimization approach. However, the Lagrange relaxation method has scaled down dynamic programming and various gradient methods as a part of it. In this case, dynamic programming is not a problem. Gradient methods pose serious problems for cost functions since the cost function is usually discontinuous. This is especially true for a unit's operational state described with only zero or one. Following the notation

and discussion of the Lagrange Relaxation method in [37], the constraints are:

$$U_i^t = \begin{cases} 0 & \text{if unit } i \text{ is off–line during period } t \\ 1 & \text{if unit } i \text{ is on–line during period } t \end{cases} \tag{C.4}$$

$$P_{load}^t - \sum_{i=1}^{N} P_i^t \; U_i^t = 0 \text{for } t=1,\ldots,\text{T} \tag{C.5}$$

$$U_i^t \; P_i^{min} \leq P_t^i \leq U_i^t \; P_i^{max} \text{for i=1,}\ldots\text{,N and t=1,}\ldots\text{,T} \tag{C.6}$$

and the objective function is:

$$\sum_{t=1}^{T} \sum_{i=1}^{N} [F_i(P_i^t) \times \mathit{Start\ up\ cost}_{i,t}] \, U_i^t = F(P_i^t, U_i^t) \tag{C.7}$$

Equations C.5, C.6, and C.7 result in the following Lagrange equation:

$$L(P, U, \lambda) = F(P_i^t, U_i^t) + \sum_{t=1}^{T} \lambda^t \left(P_{load}^t - \sum_{i=i}^{N} P_i^t \; U_i^t \right) \tag{C.8}$$

Where P_i^t is generator i output at time t, F_i is the production cost function of the generator P_i and λ is the Lagrange multiplier.

Equation C.8 has to be minimized for the unit commitment solution. Constraint C.6 is separable over the units while constraint C.5 is not. Lagrange Relaxation technique *relaxes* the constraints by temporarily ignoring the non–separable constraint and solving

the rest of the problem using the dual optimization procedure. For this purpose

$$q^*(\lambda) = \max_{\lambda^t} q(\lambda) \tag{C.9}$$

and

$$q(\lambda) = \min_{P_i^t, U_i^t} L(P, U, \lambda) \tag{C.10}$$

As formulated, this problem is manageable. $q^*(\lambda)$ is maximized using standard gradient techniques, or if gradient techniques are not possible due to the nature of the Lagrange function, some kind of a search procedure could be used. However, in this case λ is a vector. A number of local minima are possible and even convergence is questionable. $q(\lambda)$ in this form is decoupled meaning that the minimization can be done in terms of each P_i^t individually. U_i^t is accounted for by using a single variable dynamic programming. Since U_i^t can be zero or one only, the dynamic programming solution is simple. This iterative process is terminated when the duality gap is close or equal to zero. The duality gap is defined as:

$$\frac{J^* - q^*}{q^*} \tag{C.11}$$

where J^* is the primal value or the minimal value of the Lagrangian function.

There are a couple of problems with this method. Although the duality gap usually becomes small for large size systems, the process might never come to a stable ending. This is an indication that one or more generators are switched in and out between the iterative steps. The second problem is that there is no guarentee that the process will

converge to a feasible solution. Due to these problems, this method will not be discussed any further and a completely different optimization approach will be introduced in the next chapters.

C.1.4 Unit Commitment in the Deregulated Market

The unit commitment is a nontrivial optimization problem. However, with the introduction of the deregulated markets, the unit commitment must be done by all unbundled entities separately. This makes the optimization problem simpler since each entity involved must consider only the issues of its own interest. Producer issues are addressed in [5] regarding profits maximization and cost reduction considering market uncertainties over finite and infinite time horizons. Power producers must take into account start–up and shut–down costs, probable demand as well as the projected power price on the open market and almost completely ignore technical issues of the entire power system or the area to which they belong, just to mention a few issues. ISOs must address the same problem from a different point of view. ISOs are not concerned with the unit shut–down and start–up costs and such as used to be the case under the regulated power market. The ISOs concerns are system reliability, transmission control and its influence on power pricing, and price of the power supplied to the end users. Transmission owners must consider projected long term unit commitment and power demand as well as installation of new units to provide adequate transmission facilities.

It is clear that with the unbundling of the functionality, responsibilities and operational constraints are also unbundled. This will very likely make old optimization tools

obsolete. New, separate optimization tools are needed for each party involved. There are some opinions in the power system community that an ISO would not need to do unit commitment in the deregulated market and rely on the market prices. As indicated above, we do not support this opinion since the price is not the only variable that should influence ISO's decisions. It would be possible to provide a fictitious system and situation based on deregulation and market laws which could lead a price only decisions to a system collapse due to congestions and artificial power shortages.

Appendix D

Programs Code

This Appendix contains the programs used to produce the examples. Genetic Algorithm OPF and Decision Tree inducer programs are completely included while the section on Genetic Algorithm OPF based Unit commitment contains only the files which differ from the files used for Genetic Algorithm OPF.

Each section starts with Makefile and dependency files. These files can be used to determine program building, rules and library and source dependencies. In general, programs depend on GALIB and IMSL libraries. GALIB is public domain while IMSL is a commercial product.

D.1 Genetic Algorithm OPF

```
# Makefile for GAOPF
# Copyright (C) 2000
# Makefile -- Jovan Ilic ilic@thetausa.com
# Fri Apr 21 2000
# $Log$
#

.PHONY : clean

O=.o
E=
CXX=g++
CX=gcc
CXXFLAGS=-O5 -mpentiumpro
CXFLAGS=-O5 -mpentiumpro
INCD=-I/usr/local/vni/CTT3.0/include \
-I$(HOME)/include \
-I$(HOME)/ilic/include/stlport \
-I/usr/local/include
LIBD=-L$(HOME)/lib -L/usr/local/lib
LIBS=$(LINK_CNL_SHARED)  -lstlport_gcc  -lm -lga

# for Windows_NT move all LIB and
# INCLUDE variables including all
# the ones from VCVARS32.BAT to registry
ifeq ($(OS), Windows_NT)
O=.obj
E=.exe
CXX=cl
CX=cl
CXXFLAGS=/nologo /TP /GR /MT /G6  /GX \
/O2 /D "WIN32"
CXFLAGS=/nologo /G6 /MT /O2 /D "WIN32"
INCD=
LIBD=/link /nodefaultlib:libc.lib \
/nologo /subsystem:console \
/incremental:no
LIBS=dspgen.obj cmath_s.lib ga.lib \
kernel32.lib user32.lib
endif

.SUFFIXES: .cpp .c .CPP .C .CC .CXX $(O)

%$(E) : %.cpp
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.cc
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.C
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.CC
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.CXX
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(O) : %.CXX
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(O) : %.cpp
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(O) : %.cc
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(O) : %.C
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(O) : %.CC
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(E) : %.c
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) \
$(LIBS) -o $@ $<

%$(O) : %.c
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

# insert source files here

CXXSRC=main.cpp Net.cpp GetOpt.cpp options.cpp \
Bus.cpp Generator.cpp  Branch.cpp \
```

```
MatrixSPD.cpp MakeGB.cpp MakeJac.cpp  \
PFSolve.cpp LineFlows.cpp PowerCost.cpp \
ProdCost.cpp

CSRC=

OBJ := $(patsubst %.c, %$(O), $(CSRC)) $(patsubst \
%.cpp, %$(O), $(CXXSRC))


# define the target(s)

all : ./data/opf$(E)

./data/opf$(E) : $(OBJ)
$(CXX) -o $@ /home/ilic/lib/dspgen.o \
$(OBJ) $(LIBD) $(LIBS)

clean:
rm $(OBJ) ./data/opf$(E)

echo :
@echo $(OBJ)

depend :
@gcc -MM $(INCD) $(CSRC) $(CXXSRC) > depend.inc

# this file needs to exist before running make.
# "touch depend.inc" will do it
include depend.inc

# DO NOT DELETE


main.o: main.cpp stds.hpp GetOpt.h externs.hpp \
Net.hpp Bus.hpp  Generator.hpp Branch.hpp \
 MatrixSPD.hpp
Net.o: Net.cpp Net.hpp stds.hpp Bus.hpp \
Generator.hpp Branch.hpp \
  MatrixSPD.hpp externs.hpp
GetOpt.o: GetOpt.cpp GetOpt.h
options.o: options.cpp stds.hpp \
GetOpt.h externs.hpp
Bus.o: Bus.cpp Bus.hpp stds.hpp \
Generator.hpp Branch.hpp
Generator.o: Generator.cpp Generator.hpp stds.hpp
Branch.o: Branch.cpp Branch.hpp stds.hpp
MatrixSPD.o: MatrixSPD.cpp MatrixSPD.hpp
MakeGB.o: MakeGB.cpp Net.hpp stds.hpp Bus.hpp \
Generator.hpp Branch.hpp \
  MatrixSPD.hpp externs.hpp
MakeJac.o: MakeJac.cpp Net.hpp stds.hpp Bus.hpp \
Generator.hpp  Branch.hpp MatrixSPD.hpp \
externs.hpp
PFSolve.o: PFSolve.cpp Net.hpp stds.hpp Bus.hpp \
Generator.hpp  Branch.hpp MatrixSPD.hpp \
externs.hpp
LineFlows.o: LineFlows.cpp Net.hpp stds.hpp \
Bus.hpp Generator.hpp  Branch.hpp \
MatrixSPD.hpp externs.hpp
PowerCost.o: PowerCost.cpp Net.hpp stds.hpp \
Bus.hpp Generator.hpp \
  Branch.hpp MatrixSPD.hpp externs.hpp
ProdCost.o: ProdCost.cpp Net.hpp stds.hpp Bus.hpp \
Generator.hpp \
  Branch.hpp MatrixSPD.hpp externs.hpp


/*
 * - J.Ilic c:/ilic/inferno/test.h
 * Sat Jan 16 1993
 * <ilic@hickory.engr.utk.edu>
 * $Log: getopt.h $
 * Revision 0.0 1993/01/24 22:04:56 ilic
 * This is GNU file. Observe the GNU copyright!
 *
*/

#ifndef _GetOpt_h
#define _GetOpt_h 1
#include <stdio.h>
#include <string.h>

#if defined(WIN32)
#include <malloc.h>
#else
void *alloca(size_t);
#endif
```

```
#include <stdlib.h>

class GetOpt
{
private:
  /* The next char to be scanned in the option-element
     in which the last option character we returned was found.
     This allows us to pick up the scan where we left off.

     If this is zero, or a null string, it means resume the scan
     by advancing to the next ARGV-element. */

  char *nextchar;


  /* Describe how to deal with options that follow non-option
ARGV-elements.

     UNSPECIFIED means the caller did not specify anything;
     the default is then REQUIRE_ORDER if the environment variable
     _OPTIONS_FIRST is defined, PERMUTE otherwise.

     REQUIRE_ORDER means don't recognize them as options.
     Stop option processing when the first non-option is seen.
     This is what Unix does.

     PERMUTE is the default. We permute the contents of 'argv' as we
scan,
     so that eventually all the options are at the end. This allows options
     to be given in any order, even with programs that were not written to
     expect this.

     RETURN_IN_ORDER is an option available to programs that were
written
     to expect options and other ARGV-elements in any order and that
care about
     the ordering of the two. We describe each non-option ARGV-element
     as if it were the argument of an option with character code zero.
     Using '-' as the first character of the list of option characters
     requests this mode of operation.

     The special argument '--' forces an end of option-scanning regardless
     of the value of 'ordering'. In the case of RETURN_IN_ORDER, only
     '--' can cause 'getopt' to return EOF with 'optind' != ARGC. */

  enum { REQUIRE_ORDER, PERMUTE, RETURN_IN_ORDER }
ordering;

  /* Handle permutation of arguments. */

  /* Describe the part of ARGV that contains non-options that have
     been skipped. 'first_nonopt' is the index in ARGV of the first of them;
     'last_nonopt' is the index after the last of them. */

  int first_nonopt;
  int last_nonopt;

  void exchange (char **argv);
public:
  /* For communication from 'getopt' to the caller.
     When 'getopt' finds an option that takes an argument,
     the argument value is returned here.
     Also, when 'ordering' is RETURN_IN_ORDER,
     each non-option ARGV-element is returned here. */

  char *optarg;

  /* Index in ARGV of the next element to be scanned.
     This is used for communication to and from the caller
     and for communication between successive calls to 'getopt'.
     On entry to 'getopt', zero means this is the first call; initialize.

     When 'getopt' returns EOF, this is the index of the first of the
     non-option elements that the caller should itself scan.

     Otherwise, 'optind' communicates from one call to the next
     how much of ARGV has been scanned so far. */

  int optind;

  /* Callers store zero here to inhibit the error message
     for unrecognized options. */

  int opterr;
```

```
  int nargc;
  char **nargv;
  char *noptstring;

  GetOpt (int argc, char **argv, char *optstring);
  int operator () (void);
};

#endif
// -*- C++ -*-
/* Branch.hpp - Jovan Ilic ilic@thetausa.com Tue Sep 5 2000
 *
 * $Header$
 * $Log$
 */

#ifndef _BRANCH_HPP
#define _BRANCH_HPP

#include "stds.hpp"

class Branch {
private: // make clear that private part is first by default
  int brFB, brTB, brTapBus, brNonTapBus;
  double brR, brX, brBcap, brLineFlowLimit, brTapRatio;
  string brLineType;

protected: // protected members

public: // public members
  Branch(){ // default constructor
    brFB = brTB = brTapBus = brNonTapBus = -1;
    brR = brX = brBcap = brLineFlowLimit = brTapRatio = 1.0;
    brLineType = "noType";
  }

  Branch(const Branch &b){ // copy constructor
    brFB = b.BrFB();
    brTB = b.BrTB();
    brTapBus = b.BrTapBus();
    brNonTapBus = b.BrNonTapBus();
    brR = b.BrR();
    brX = b.BrX();
    brBcap = b.BrBcap();
    brLineFlowLimit = b.BrLineFlowLimit();
    brTapRatio = b.BrTapRatio();
    brLineType = b.BrLineType();
  }

  Branch &operator=(const Branch &b){ // overloaded = operator
    brFB = b.BrFB();
    brTB = b.BrTB();
    brTapBus = b.BrTapBus();
    brNonTapBus = b.BrNonTapBus();
    brR = b.BrR();
    brX = b.BrX();
    brBcap = b.BrBcap();
    brLineFlowLimit = b.BrLineFlowLimit();
    brTapRatio = b.BrTapRatio();
    brLineType = b.BrLineType();
    return *this;
  }

  int BrFB() const {return brFB;}
  void BrFB(const int b) {brFB = b;}

  int BrTB() const {return brTB;}
  void BrTB(const int b) {brTB = b;}

  int BrTapBus() const {return brTapBus;}
  void BrTapBus(const int b) { brTapBus = b;}

  int BrNonTapBus() const {return brNonTapBus;}
  void BrNonTapBus(const int b) {brNonTapBus = b;}

  double BrR() const {return brR;}
  void BrR(const double r) {brR = r;}

  double BrX() const {return brX;}
  void BrX(const double x) {brX = x;}

  double BrBcap() const {return brBcap;}
  void BrBcap(const double bc) {brBcap = bc;}

  double BrLineFlowLimit() const {return brLineFlowLimit;}
  void BrLineFlowLimit(const double fl) {brLineFlowLimit = fl;}
```

```
  double BrTapRatio() const { return brTapRatio;}
  void BrTapRatio(const double tr) { brTapRatio = tr;}

  string BrLineType() const {return brLineType;}
  void BrLineType(const string lt) {brLineType = lt;}

  virtual ~Branch(){}; // virtual destructor

  friend ostream &operator<<(ostream &, const Branch &);

};

#endif
 // -*- C++ -*-
/* Bus.hpp - Jovan Ilic ilic@thetausa.com Tue Sep 5 2000
 *
 * $Header$
 * $Log$
 */

#ifndef _BUS_HPP
#define _BUS_HPP

#include "stds.hpp"

#include "Generator.hpp"
#include "Branch.hpp"

class Bus {
private: // make clear that private part is first by default

  // sync this enum with other classes
  enum {GEN = 2, LOAD, GEN_HIGHVAR, GEN_LOWVAR, GEN_REG,
SWING};

  int busNo;
  int type; // bus type from enum in case it changes.
  string busType, busName;
  double pLoad, qLoad, Vhl, Vll;
  double ySh;

  // do not change the voltage between runs for faster
  // convergence. v is in polar coordinates.
  complex<double> v;
  double pNetInj, qNetInj;
  // Power flow between adjacent buses.
  vector<complex<double> > lineFlow;

  // qVarLimit needs to be set to zero between reruns
  // as well as the bus types
  double qVarLimit;

  // attach a generator if there is one at this bus
  Generator *gen;

  // list of connecting branches
  vector<Branch *> adjBranch;

  // list of adjacent buses can be deduced from the
  // list of connecting branches but the list of
  // adjacent buses is needed so frequently that it
  // is justified to keep it in memory
  vector<int> adjBuses, adjUBus;

  // voltage penalty parameters
  double maxPenalty, rightNomV, leftNomV, dv;
  double aLeft, bLeft, cLeft; // left polynomial constants
  double aRight, bRight, cRight; // right polynomial constants

protected: // protected members

public: // public members
  Bus(){ // default constructor
    ySh = 0.0;
    busNo = -1;
    type = LOAD;
    busType = busName = "none";
    pLoad = qLoad = 0.0;
    Vhl = 1.05;
    Vll = 0.95;
    v = 1.0;
    pNetInj = qNetInj = 0.0;
    qVarLimit = 0.0;
    gen = NULL;
```

```
  // initialize this just in case
  maxPenalty = 1.0;
  leftNomV = 1.025;
  rightNomV = 0.975;
  dv = 0.025;
  InitPenaltyPolynomial();
}

Bus(const Bus &b){ // copy constructor
  ySh = b.YSh();
  busNo = b.BusNo();
  type = b.Type();
  busType = b.BusType();
  busName = b.BusName();
  pLoad = b.PLoad();
  qLoad = b.QLoad();
  Vhl = b.VHL();
  Vll = b.VLL();
  v = b.V();
  pNetInj = b.PNetInj();
  qNetInj = b.QNetInj();
  qVarLimit = b.QVarLimit();
  gen = b.Gen();

  // note: adjBranch is a vector of pointers.
  adjBranch = const_cast<Bus &>(b).AdjBranch();
  adjBuses = const_cast<Bus &>(b).AdjBus();

  maxPenalty = b.MaxPenalty();
  leftNomV = b.LeftNomV();
  rightNomV = b.RightNomV();
  dv = b.Dv();
  InitPenaltyPolynomial();
}

Bus &operator=(const Bus &b){ // overloaded = operator
  ySh = b.YSh();
  busNo = b.BusNo();
  type = b.Type();
  busType = b.BusType();
  busName = b.BusName();
  pLoad = b.PLoad();
  qLoad = b.QLoad();
  Vhl = b.VHL();
  Vll = b.VLL();
  v = b.V();
  pNetInj = b.PNetInj();
  qNetInj = b.QNetInj();
  qVarLimit = b.QVarLimit();
  gen = b.Gen();

  // note: adjBranch is a vector of pointers
  adjBranch = const_cast<Bus &>(b).AdjBranch();
  adjBuses = const_cast<Bus &>(b).AdjBus();

  maxPenalty = b.MaxPenalty();
  leftNomV = b.LeftNomV();
  rightNomV = b.RightNomV();
  dv = b.Dv();
  InitPenaltyPolynomial();

  return *this;
}

virtual ~Bus(){}; // virtual destructor

double YSh() const { return ySh; }
void YSh(const double ysh) { ySh = ysh; }

int BusNo() const {return busNo;}
void BusNo(const int bn) {busNo = bn;}

string BusType() const {return busType;}
void BusType(const string bt) { busType = bt;}

int Type() const { return type;}
void Type(const int t) {
  if((t >= GEN) && (t <= SWING))
    type = t;
}

string BusName() const {return busName;}
void BusName(const string bn) {busName = bn;}

double PLoad() const {return pLoad;}
void PLoad(const double pl) { pLoad = pl;}
```

```
double QLoad() const {return qLoad;}
void QLoad(const double ql) {qLoad = ql;}

double QGenMin() {
  if(gen != NULL)
    return gen→GenQMin();
  else
    return qNetInj;
}

double QGenMax() {
  if(gen != NULL)
    return gen→GenQMax();
  else
    return qNetInj;
}

double PGen() const {
  if( gen != NULL)
    return gen→GenP();
  else
    return 0;
}

double QGen() const { return (qNetInj + qLoad); }

double VHL() const {return Vhl;}
void VHL(const double vhl) { Vhl = vhl;}

double VLL() const {return Vll;}
void VLL(const double vll) {Vll = vll;}

double PNetInj() const { return pNetInj;}
void PNetInj(const double p) { pNetInj = p; }

double QNetInj() const { return qNetInj;}
void QNetInj(const double q) { qNetInj = q;}

Generator *Gen() const { return gen;}
void Gen(const Generator *g) {
  gen = const_cast<Generator *>(g);
}
```

```
double PSched() {
  if(gen != NULL)
    return(gen→GenP() - pLoad);
  else
    return(-pLoad);
}

double QSched() {
  return(qVarLimit - qLoad);
}

complex<double> V() const {return v;}
void V(const complex<double> vc) { v = vc;}

complex<double> GenV() const {
  // Bus not the Generator carries the actual bus type.
  // Generator carries the intended/original bus type value.
  if((type == SWING) || (type == GEN))
    return gen→GenV();
  else
    return v;
}

void AdjBranch(Branch *br) { adjBranch.push_back(br); }
const vector<Branch *> &AdjBranch() { return adjBranch; }

void AdjBus(int bs) { adjBuses.push_back(bs); }
const vector<int> &AdjBus() { return adjBuses; }

void AdjUBus(int ubs) {adjUBus.push_back(ubs); }
const vector<int> &AdjUBus() { return adjUBus; }
void UniqueAdjBus();

double QVarLimit() const { return qVarLimit; }
void QVarLimit(const double qv) { qVarLimit = qv;}

vector<complex<double> > &LineFlow() { return lineFlow;}

double MaxPenalty() const { return maxPenalty;}
void MaxPenalty(const double mp) { maxPenalty = mp;}

double LeftNomV() const {return leftNomV;}
```

```
  void LeftNomV(const double lf) {leftNomV = lf;}

  double RightNomV() const {return rightNomV;}
  void RightNomV(const double rg) {rightNomV = rg;}

  double Dv() const {return dv;}
  void Dv(const double rd) { dv = rd;}

  void InitPenaltyPolynomial();
  double VoltagePenalty();

  friend ostream &operator≪(ostream &, const Bus &);
};

#endif
// -*- C++ -*-
/* externs.hpp - Jovan Ilic ilic@thetausa.com Mon Oct 4 1999
 *
 * $Header$
 * $Log$
 */

#ifndef _EXTERNS_HPP
#define _EXTERNS_HPP

extern string InputFile_str, OutGA_str, OutTol_str;
extern string GAParms_str, SysOut_str, ErrsGA_str;

extern int interval, continuous, iZero;

extern void GetOptionsHelp(int, char **);
extern void GetOptions(int, char **);

// Microsoft VC++ specific
#ifdef _MSC_VER

#define M_PI 3.14159265358979323846

#endif // _MSC_VER


#endif
// -*- C++ -*-
```

```
/* Generator.hpp - Jovan Ilic ilic@thetausa.com Tue Sep 5 2000
 *
 * $Header$
 * $Log$
 */

#ifndef _GENERATOR_HPP
#define _GENERATOR_HPP

#include "stds.hpp"

#include <math.h>

class Generator {
private: // make clear that private part is first by default
  enum {GEN = 2, LOAD, GEN_HIGHVAR, GEN_LOWVAR, GEN_REG,
SWING};
  int genBusNo, tp;
  double genP, genV, genQMax, genQMin;

  // cost function calculation additions
  // This could also be a line segment description
  vector<double> costPoly; // poly coefficients in increasing order
  double genPMin, genPMax; // generation limits

protected: // protected members

public: // public members

  Generator(){ // default constructor
    genBusNo = -1;
    tp = GEN;
    genP = genV = genQMax = genQMin = 0.0;
    genPMin = genPMax = 0.0;

  }

  Generator(const Generator &g){ // copy constructor
    genBusNo = g.GenBusNo();
    tp = g.Type();
    genP = g.GenP();
    genV = g.GenV();
    genQMax = g.GenQMax();
```

```
  genQMin = g.GenQMin();
  genPMin = g.GenPMin();
  genPMax = g.GenPMax();
  costPoly = const_cast<Generator &>(g).CostPolyV();

}

Generator &operator=(const Generator &g){ // overloaded = operator
  genBusNo = g.GenBusNo();
  tp = g.Type();
  genP = g.GenP();
  genV = g.GenV();
  genQMax = g.GenQMax();
  genQMin = g.GenQMin();
  genPMin = g.GenPMin();
  genPMax = g.GenPMax();
  costPoly = const_cast<Generator &>(g).CostPolyV();

  return *this;
}

virtual ~Generator(){}; // virtual destructor

int GenBusNo() const {return genBusNo;}
void GenBusNo(const int g) {genBusNo = g;}

double GenP() const { return genP;}
void GenP(const double gp) {genP = gp;}

double GenV() const { return genV;}
void GenV(const double gv) { genV = gv;}

double GenQMax() const {return genQMax; }
void GenQMax(const double gq) {genQMax = gq;}

double GenQMin() const { return genQMin;}
void GenQMin(const double gq) { genQMin = gq; }

double GenPMax() const { return genPMax;}
void GenPMax(const double gp) { genPMax = gp;}

double GenPMin() const {return genPMin;}
void GenPMin(const double gp) {genPMin = gp;}

// save (in order) a polynomial cost coefficient
void CostPoly(const double coef) {
  costPoly.push_back(coef);
}

// whole cost vector assignment and retrival
const vector<double> &CostPolyV() { return costPoly; }
void CostPoly(const vector<double> &c) { costPoly = c;}

// for Cost() and CostInc(), argumment should be between
// genPMin and genPMax. p should be non-pu power
// genPMin and genPMax are also not scaled
double Cost(double p) {
  double c = 0.0;
  // generation limits are here only because of the swing bus.
  if(p != 0.0) {
    if(p < genPMin)
return 2.0*Cost(genPMax);

    if(p > genPMax)
return 4*Cost(genPMax);
  }

  for(int i = 0; i < costPoly.size(); i++)
    c += costPoly[i] * pow(p, static_cast<double>(i));
  return c;
}

double CostInc(double p) {
  double c = 0.0;
  if(p < genPMin)
    return CostInc(genPMin);
  if(p > genPMax)
    return CostInc(genPMax);

  for(int i = 1; i < costPoly.size(); i++)
    c += costPoly[i] * i * pow(p, static_cast<double>(i - 1));
  return c;
}


int Type() const { return tp;}
```

```
  void Type(const int t) {
    if((t <= SWING) && (t >= GEN))
      tp = t;
  }

  friend ostream &operator≪(ostream &, const Generator &);

};

#endif
// -*- C++ -*-
/* MatrixSPD.hpp - Jovan Ilic ilic@thetausa.com Fri Dec 29 2000
 *
 * $Header$
 * $Log$
 */

#ifndef _MATRIXSPD_HPP
#define _MATRIXSPD_HPP


#include <iostream>
#include <vector>
#include <map>
#include <imsl.h>


#if defined(__SGI_STL_PORT)
using namespace _STL;
#else
using namespace std;
#endif

// This type definitions match SuperLU enums
typedef enum {
  GE, /* general */
  TRLU, /* lower triangular, unit diagonal */
  TRUU, /* upper triangular, unit diagonal */
  TRL, /* lower triangular */
  TRU, /* upper triangular */
  SYL, /* symmetric, store lower half */
  SYU, /* symmetric, store upper half */
  HEL, /* Hermitian, store lower half */
  HEU /* Hermitian, store upper half */
} Mtype_t;

typedef enum {
  _S, /* single */
  _D, /* double */
  _C, /* single complex */
  _Z /* double complex */
} Dtype_t;

class MatrixSPD {
private: // make clear that private part is first by default

  // vector of columns makes it easier to
  //switch to Harwell-Boeing data format
  vector<map<int, double, less<int> > > dat;

  // HBvalid is false whenever HB arrays need to be
  // (re)built.
  bool HBvalid;

  // matrix format type for use with SuperLU
  Mtype_t atype;

  // Harwell-Boeing arrays kept in STL vectors
  vector<double> a;
  vector<int> asub, xa;
  // number of rows, columns, and nonzero elements
  int m, n, nnz;

  void UpdateHB();

protected: // protected members


public: // public members

  // need to handle index errors somehow
  // Using exceptions would be the right way - maybe
  double err;

  MatrixSPD(); // default constructor
```

```
  MatrixSPD(int, int); // constructor with dimensions
  MatrixSPD(int, int, int);

  MatrixSPD(const MatrixSPD &); // copy constructor

  MatrixSPD &operator=(const MatrixSPD &); // overloaded =
operator

  virtual ~MatrixSPD(){}; // virtual destructor (pure ??)

  Mtype_t Atype() const { return atype;}
  void Atype(const Mtype_t mt) { atype = mt;}

  const vector<double> &A() const { return a;}
  void A(const vector<double> &aa) { a = const_cast<vector<double>
&>(aa); }

  const vector<int> &Asub() const { return asub; }
  void Asub(const vector<int> &as) { asub = const_cast<vector<int>
&>(as);}

  const vector<int> &Xa() const { return xa;}
  void Xa(const vector<int> &x) { xa = const_cast<vector<int> &>(x);
}

  int M() const {return m;}
  void M(const int ms) { m = ms;}

  int N() const {return n;}
  void N(const int ns) { n = ns;}

  int NNZ() const { return nnz; }
  // here we might have to adjust internal vectors
  void NNZ(const int nz){nnz = nz;}

  bool SetA(int,int,int,vector<double> &,vector<int> &,vector<int>
&,Mtype_t);
  bool SetA(vector<double> &, vector<int> &, vector<int> &,
Mtype_t, int = 0);

  // access HB column maps
  const map<int, double, less<int> > &Col(int) const;
  void Col(const map<int, double, less<int> > &, int);

  bool HBValid() const;
  void HBValid(const bool);

  void PrintMap(ostream &);
  void PrintHB(ostream &);

  friend ostream &operator<<(ostream &, const MatrixSPD &);

  // might want to overload this member a few times
  bool Solve(vector<double> &, vector<double> &);

  double Aij(int, int) const ;
  double Aij(const int, const int, const double);

};

#endif
// -*- C++ -*-
/* Net.hpp - Jovan Ilic ilic@thetausa.com Tue Sep 5 2000
 *
 * $Header$
 * $Log$
 */

#ifndef _NET_HPP
#define _NET_HPP

#include "stds.hpp"

#include "Bus.hpp"
#include "Generator.hpp"
#include "Branch.hpp"
#include "MatrixSPD.hpp"

#include "externs.hpp"

class Net {
private: // make clear that private part is first by default

  enum {LIN, TRAN, GEN, LOAD, GEN_HIGHVAR, GEN_LOWVAR,
GEN_REG, SWING};
```

```
double pBase;

int noBus; // number of buses from data file
int noGen; // number of generators from data file
int noBr; // number of branches from data file
int swingBus; // swing bus index from data file
int noTran; // number of transformers
int noCap; // number of buses with capacitors

// title lines from data file
string Title1, Title2;

// power flow limiting variables
double pMaxMismatch, qMaxMismatch;
double pMismatch, qMismatch;
int maxIter;

// used for SuperLU state
char fact;

// handling generator var limits, false means no limits violated.
bool varLimit;

vector<Bus> buses;
vector<Generator> gens;
vector<Branch> branches;
// Transformers and capacitors indecies.
vector<int> trans, caps;

// sparse admittance matrix vectors
MatrixSPD G, B;
vector<double> g, b;
vector<int> gsub, xg;
vector<int> bsub, xb;

// sparse Jacobian matrix vectors
MatrixSPD J;
vector<double> jac;
vector<int> jacsub, xjac;

// additional variables used in power flow calculations
vector<double> delPQ;

// variables needed to maintain correct Jacobian format
int PQCount, PVCount, VCCount;
vector<int> rowShifts;

// output variables
double pLoadTotal, qLoadTotal, pGenTotal, qGenTotal;
double totalI2R, totalI2X, totalCharging;

// Power Injection based on the currently calculated voltages
bool NetInj();

// Difference between the scheduled and calculated power injections
bool DelPQ();

// Checks generators for exceeding VAR limits and makes bus VAR
adjustments
bool CheckVARLimits();


protected: // protected members


public: // public members

Net(){ // default constructor
  pBase = 100.0;
  noBus = noGen = noBr = swingBus = noTran = noCap = -1;
  Title1 = Title2 = "No Comment";
  pMaxMismatch = qMaxMismatch = 0.0001;
  pMismatch = qMismatch = 0.0;
  maxIter = 50;
  fact = 'N';
  varLimit = false;
  PQCount = PVCount = VCCount = 0;
  pLoadTotal = qLoadTotal = pGenTotal = qGenTotal = 0.0;
  totalI2R = totalI2X = totalCharging = 0.0;
}

Net(const Net &n){ // copy constructor
  pBase = n.PBase();
  noBus = n.NoBus();
  noGen = n.NoGen();
  noBr = n.NoBr();
```

```
swingBus = n.SwingBus();
noTran = n.NoTran();
noCap = n.NoCap();
Title1 = n.title1();
Title2 = n.title2();
pMaxMismatch = n.PMaxMismatch();
qMaxMismatch = n.QMaxMismatch();
pMismatch = n.PMismatch();
qMismatch = n.QMismatch();
maxIter = n.MaxIter();
fact = n.Fact();
varLimit = n.VarLimit();
PQCount = n.pQCount();
PVCount = n.pVCount();
VCCount = n.vCCount();

buses = const_cast<Net &>(n).BusesV();
gens = const_cast<Net &>(n).GensV();
branches = const_cast<Net &>(n).BranchesV();

// G and B parts of the admittance matrix
g = const_cast<Net &>(n).GV();
b = const_cast<Net &>(n).BV();
gsub = const_cast<Net &>(n).GsubV();
xg = const_cast<Net &>(n).XgV();
bsub = const_cast<Net &>(n).GsubV();
xb = const_cast<Net &>(n).XbV();

// make DSuperLU objects for G and B sparse matrices
G.SetA(g, gsub, xg, SYL);
B.SetA(b, bsub, xb, SYL);

// Jacobian sparse matrix
jac = const_cast<Net &>(n).JacV();
jacsub = const_cast<Net &>(n).JacsubV();
xjac = const_cast<Net &>(n).XjacV();

// make DSuperLU object for Jacobian
J.SetA(jac, jacsub, xjac, GE);

delPQ = const_cast<Net &>(n).DelPQV();

rowShifts = const_cast<Net &>(n).RowShiftsV();
```



```
  pLoadTotal = PLoadTotal();
  qLoadTotal = QLoadTotal();
  pGenTotal = PGenTotal();
  qGenTotal = QGenTotal();
  totalI2R = TotalI2R();
  totalI2X = TotalI2X();
  totalCharging = TotalCharging();

}

Net &operator=(const Net &n){ // overloaded = operator

  pBase = n.PBase();
  noBus = n.NoBus();
  noGen = n.NoGen();
  noBr = n.NoBr();
  swingBus = n.SwingBus();
  noTran = n.NoTran();
  noCap = n.NoCap();
  Title1 = n.title1();
  Title2 = n.title2();
  pMaxMismatch = n.PMaxMismatch();
  qMaxMismatch = n.QMaxMismatch();
  pMismatch = n.PMismatch();
  qMismatch = n.QMismatch();
  maxIter = n.MaxIter();
  fact = n.Fact();
  varLimit = n.VarLimit();
  PQCount = n.pQCount();
  PVCount = n.pVCount();
  VCCount = n.vCCount();

  buses = const_cast<Net &>(n).BusesV();
  gens = const_cast<Net &>(n).GensV();
  branches = const_cast<Net &>(n).BranchesV();

  // G and B parts of the admittance matrix
  g = const_cast<Net &>(n).GV();
  b = const_cast<Net &>(n).BV();
  gsub = const_cast<Net &>(n).GsubV();
  xg = const_cast<Net &>(n).XgV();
  bsub = const_cast<Net &>(n).GsubV();
```

```
    xb = const_cast<Net &>(n).XbV();

    // make DSuperLU objects for G and B sparse matrices
    G.SetA(g, gsub, xg, SYL);
    B.SetA(b, bsub, xb, SYL);

    // Jacobian sparse matrix
    jac = const_cast<Net &>(n).JacV();
    jacsub = const_cast<Net &>(n).JacsubV();
    xjac = const_cast<Net &>(n).XjacV();

    // make DSuperLU object for Jacobian
    J.SetA(jac, jacsub, xjac, GE);

    delPQ = const_cast<Net &>(n).DelPQV();

    rowShifts = const_cast<Net &>(n).RowShiftsV();

    pLoadTotal = PLoadTotal();
    qLoadTotal = QLoadTotal();
    pGenTotal = PGenTotal();
    qGenTotal = QGenTotal();
    totalI2R = TotalI2R();
    totalI2X = TotalI2X();
    totalCharging = TotalCharging();
    return *this;
}

virtual ~Net(){}; // virtual destructor

double PBase() const {return pBase;}
void PBase(const double pb) {pBase = pb;}

int NoBus() const { return noBus; }
void NoBus(const int nb) { noBus = nb; }

int NoGen() const { return noGen; }
void NoGen(const int ng) { noGen = ng; }

int NoBr() const {return noBr; }
void NoBr(const int nb) { noBr = nb; }

int SwingBus() const { return swingBus; }
void SwingBus(const int sw) { swingBus = sw; }

int NoTran() const { return noTran; }
void NoTran(const int notr) { noTran = notr; }

int NoCap() const { return noCap; }
void NoCap(const int nc) { noCap = nc; }

string title1() const { return Title1; }
void title1(const string tl) { Title1 = tl; }

string title2() const { return Title2; }
void title2(const string tl) { Title2 = tl; }

double PMaxMismatch() const { return pMaxMismatch; }
void PMaxMismatch(const double mm) { pMaxMismatch = mm;}

double QMaxMismatch() const { return qMaxMismatch; }
void QMaxMismatch(const double mm) { qMaxMismatch = mm;}

double PMismatch() const { return pMismatch; }
void PMismatch(const double pm) { pMismatch = pm; }

double QMismatch() const { return qMismatch; }
void QMismatch(const double qm) { qMismatch = qm; }

int MaxIter() const { return maxIter;}
void MaxIter(const int mi) { maxIter = mi; }

char Fact() const { return fact; }
void Fact(const char f) { fact = f; }

bool VarLimit() const { return varLimit; }
void VarLimit(const bool vl) { varLimit = vl; }

int pQCount() const { return PQCount; }
void pQCount(const int pqc) { PQCount = pqc; }

int pVCount() const { return PVCount; }
void pVCount(const int pvc) { PVCount = pvc; }

int vCCount() const { return VCCount; }
void vCCount(const int vcc) { VCCount = vcc; }
```

```
const vector<Bus> &BusesV() const { return buses; }
void BusesV(const vector<Bus> &b) { buses = b; }

const vector<Generator> &GensV() const { return gens; }
void GensV(const vector<Generator> &g) { gens = g; }

const vector<Branch> &BranchesV() const { return branches; }
void BranchesV(const vector<Branch> &b) { branches = b;}

const vector<double> &GV() const { return g;}
void GV(const vector<double> &gv) { g = gv; }

const vector<double> &BV() const { return b;}
void BV(const vector<double> &bv) { b = bv;}

const vector<int> &GsubV() const { return gsub;}
void GsubV(const vector<int> &gg) { gsub = gg; }

const vector<int> &XgV() const { return xg;}
void XgV(const vector<int> &gg) { xg = gg; }

const vector<int> &BsubV() const { return bsub; }
void BsubV(const vector<int> &gg) { bsub = gg; }

const vector<int> XbV() const { return xb; }
void XbV(const vector<int> &gg) { xb = gg; }

const vector<double> &JacV() const { return jac; }
void JacV(const vector<double> &j) { jac = j;}

const vector<int> &JacsubV() const { return jacsub; }
void JacsubV(const vector<int> &j) { jacsub = j; }

const vector<int> &XjacV() const { return xjac; }
void XjacV(const vector<int> &xj) { xjac = xj; }

const vector<double> &DelPQV() const { return delPQ; }
void DelPQV(const vector<double> &dpq) { delPQ = dpq; }

const vector<int> &RowShiftsV() const { return rowShifts; }
void RowShifts(const vector<int> &rs) { rowShifts = rs; }
```

```
double PLoadTotal() const {return pLoadTotal; }
void PLoadTotal(const double pt) { pLoadTotal = pt; }

double QLoadTotal() const { return qLoadTotal; }
void QloadTotal(const double qt) { qLoadTotal = qt; }

double PGenTotal() const { return pGenTotal; }
void PGenTotal(const double pg) { pGenTotal = pg;}

double QGenTotal() const { return qGenTotal;}
void QGenTotal(const double qg) { qGenTotal = qg;}

double TotalI2R() const { return totalI2R; }
void TotalI2R(const double tr) { totalI2R = tr;}

double TotalI2X() const { return totalI2X; }
void TotalI2X(const double tx) { totalI2X = tx;}

double TotalCharging() const { return totalCharging; }
void TotalCharging(const double tc) { totalCharging = tc; }

bool ReadAW(const string &);
bool PrintAW(ostream &);
bool PrintAWOut(ostream &os);
bool MakeY();
bool MakeJ();

// Jacobian retains the same format but it needs to be
// recalculated when solving the equations
bool ReCalcJ();

bool ResetBuses();

const vector<Bus> &Buses() const { return buses;}

bool PFSolve();

bool LineFlows();

double ProdCost();

double ProdCostScaled();
```

```
  double VoltagePenalty(){
    double sump = 0.0;
    for(int i = 0; i < buses.size(); i++)
      sump += buses[i].VoltagePenalty();
    return sump;
  }

};

#endif
// -*- C++ -*-
/* stds.hpp - Jovan Ilic ilic@thetausa.com Mon May 08 2000
 *
 * $Header$
 * $Log$
 */

#ifndef _STDS_HPP
#define _STDS_HPP

#include <iostream>
#include <strstream>
#include <iomanip>
#include <string>
#include <stack>
#include <fstream>
#include <vector>
#include <map>
#include <algorithm>
#include <iterator>
#include <ctime>
#include <complex>


#include <stdlib.h>

#if defined(_SGI_STL_PORT)
using namespace _STL;
#else
using namespace std;
#endif

#if defined(WIN32)
using std::setw;
using std::setprecision;
using std::vector;
using std::cout;
using std::cerr;
using std::endl;
using std::string;
using std::fstream;
using std::map;
using std::less;
using std::ostream;
using std::stack;
using std::ios;
using std::ostream_iterator;
using std::sort;
using std::clock;
using std::strstream;
using std::complex;
#endif

#endif
// -*- C++ -*-
/* Branch.cpp - Jovan Ilic ilic@thetausa.com Tue Sep 5 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "Branch.hpp"

ostream &operator<<(ostream &os, const Branch &br) {
  ios::fmtflags oldFlags = os.flags();
  os.setf(ios::left, ios::adjustfield);
  os << setw(4) << br.brFB << " ";
  os << setw(4) << br.brTB << " ";
  os << setw(6) << setprecision(4) << br.brR << " ";
  os << setw(6) << setprecision(4) << br.brX << " ";
  os << setw(8) << setprecision(6) << br.brBcap << " ";
  os << setw(9) << setprecision(4) << br.brLineFlowLimit << " ";
  os << br.brLineType << " ";
```

```
  for(int j = 0; j < 6 - br.brLineType.size(); j++)
    os << ' ';
  if(br.brLineType == "TRAN") {
    os << setw(4) << br.brTapBus << " ";
    os << setw(7) << br.brNonTapBus << " ";
    os << setw(6) << setprecision(4) << br.brTapRatio;
  }
  os << endl;
  os.flags(oldFlags);
  return os;
}

bool lessFB(const Branch &b1, const Branch &b2) {
  return (b1.BrFB() < b2.BrFB());
}

bool lessTB(const Branch &b1, const Branch &b2) {
  return (b1.BrTB() < b2.BrTB());
}

// -*- C++ -*-
/* Bus.cpp - Jovan Ilic ilic@thetausa.com Tue Sep 5 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "Bus.hpp"

ostream &operator<<(ostream &os, const Bus &bus) {
  ios::fmtflags oldFlags = os.flags();
  os.setf(ios::left, ios::adjustfield);
  os << setw(6) << bus.BusNo() << " " << bus.BusName() << " ";
  // a little string adjustment.
  for(int j = 0; j < 9 - bus.BusName().size(); j++)
    os << ' ';
  os << setw(10) << setprecision(5) << (bus.PLoad()) << " ";
  os << setw(10) << setprecision(5) << (bus.QLoad()) << " ";
  os << setw(10) << setprecision(5) << bus.VLL() << " ";
  os << setw(10) << setprecision(5) << bus.VHL() << " ";
  os << bus.BusType() << endl;
  os.flags(oldFlags);
  return os;
}

bool lessBus(const Bus &b1, const Bus &b2) {
  return (b1.BusNo() < b2.BusNo());
}

double Bus::VoltagePenalty() {
  double vm = real(v), p;
  if(type == GEN || type == SWING || type == GEN_REG)
    return 0.0;
  if(vm <= leftNomV){
    p = (aLeft * (vm * vm) + bLeft * vm + cLeft);
    return p;
  }
  if(vm >= rightNomV){
    p = (aRight * (vm * vm) + bRight * vm + cRight);
    return p;
  }
  return 0.0;
}

void Bus::InitPenaltyPolynomial(){ // call this after the above is initialized.
  rightNomV = Vhl - dv;
  leftNomV = Vll + dv;
  aLeft = maxPenalty/(dv*dv);
  bLeft = -(2.0*maxPenalty * leftNomV)/(dv * dv);
  cLeft = (maxPenalty * leftNomV * leftNomV)/(dv * dv);
  aRight = maxPenalty/(dv*dv);
  bRight = -(2.0*maxPenalty*rightNomV)/(dv * dv);
  cRight = (maxPenalty * rightNomV * rightNomV)/(dv * dv);
}

void Bus::UniqueAdjBus() {
  adjUBus.erase(adjUBus.begin(), adjUBus.end());
  adjUBus.reserve(adjBuses.size());
  for(int i = 0; i < adjBuses.size(); i++){
    if(find(adjUBus.begin(), adjUBus.end(), adjBuses[i]) != adjUBus.end())
      continue;
    else
      adjUBus.push_back(adjBuses[i]);
```

```
  }
}
// -*- C++ -*-
/* Generator.cpp - Jovan Ilic ilic@tehtausa.com Tue Sep 5 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "Generator.hpp"


ostream &operator<<(ostream &os, const Generator &gen) {
  int i;
  ios::fmtflags oldFlags = os.flags();
  os.setf(ios::left, ios::adjustfield);
  os << setw(6) << gen.genBusNo << " ";
  os << setw(5) << setprecision(4) << gen.genP << " ";
  os << setw(6) << setprecision(5) << gen.genV << " ";
  os << setw(7) << setprecision(4) << gen.genQMax << " ";
  os << setw(7) << setprecision(4) << gen.genQMin << " ";
  os << setw(6) << setprecision(4) << gen.genPMin << " ";
  os << setw(7) << setprecision(4) << gen.genPMax << " ";

  for(i = 0; i < gen.costPoly.size(); i++)
    os << setw(7) << setprecision(4) << gen.costPoly[i] << " ";

  os << endl;
  os.flags(oldFlags);

  return os;
}

bool lessGen(const Generator &g1, const Generator &g2) {
  return (g1.GenBusNo() < g2.GenBusNo());
}
/* - J.Ilic e:/ilic/pc/inferno/options.cc
 * Wed Oct 14 1992
 * <ilic@hickory.engr.utk.edu>
 */
```



```
// $Log: getopt.cc $
// Revision 0.0 1993/01/15 19:19:51 ilic
// This is first and final version of GNU implementation
// of getopt function. Observe the GNU copyright.
//

#ifndef lint
static char *rcsid = "$Id:  getopt.cc 0.0 1993/01/15 19:19:51 ilic
Exp ilic $";
#endif

#include <stdlib.h>

#include "GetOpt.h"

GetOpt::GetOpt (int argc, char **argv, char *optstring): opterr (1)
{
  /* Initialize the internal data when the first call is made.
     Start processing options with ARGV-element 1 (since ARGV-element
0
     is the program name); the sequence of previously skipped
     non-option ARGV-elements is empty. */

  nargc = argc;
  nargv = argv;
  noptstring = optstring;
  first_nonopt = last_nonopt = optind = 1;
  optarg = nextchar = 0;

  /* Determine how to handle the ordering of options and nonoptions. */

  if (optstring[0] == '-')
    ordering = RETURN_IN_ORDER;
  else if (getenv ("_POSIX_OPTION_ORDER") != 0)
    ordering = REQUIRE_ORDER;
  else
    ordering = PERMUTE;
}
void
GetOpt::exchange (char **argv)
{
  int nonopts_size
```

```
    = (last_nonopt - first_nonopt) * sizeof (char *);
  char **temp = (char **) alloca (nonopts_size);

  /* Interchange the two blocks of data in argv. */

  memcpy (&argv[first_nonopt], temp, nonopts_size);
  memcpy (&argv[last_nonopt], &argv[first_nonopt],
          (optind - last_nonopt) * sizeof (char *));
  memcpy (temp, &argv[first_nonopt + optind - last_nonopt],
          nonopts_size);

  /* Update records for the slots the non-options now occupy. */

  first_nonopt += (optind - last_nonopt);
  last_nonopt = optind;
}

/* Scan elements of ARGV (whose length is ARGC) for option characters
   given in OPTSTRING.

   If an element of ARGV starts with '-', and is not exactly "-" or "--",
   then it is an option element.  The characters of this element
   (aside from the initial '-') are option characters.  If 'getopt'
   is called repeatedly, it returns successively each of theoption characters
   from each of the option elements.

   If 'getopt' finds another option character, it returns that character,
   updating 'optind' and 'nextchar' so that the next call to 'getopt' can
   resume the scan with the following option character or ARGV-element.

   If there are no more option characters, 'getopt' returns 'EOF'.
   Then 'optind' is the index in ARGV of the first ARGV-element
   that is not an option.  (The ARGV-elements have been permuted
   so that those that are not options now come last.)

   OPTSTRING is a string containing the legitimate option characters.
   A colon in OPTSTRING means that the previous character is an option
   that wants an argument.  The argument is taken from the rest of the
   current ARGV-element, or from the following ARGV-element,
   and returned in 'optarg'.

   If an option character is seen that is not listed in OPTSTRING,
   return '?' after printing an error message.  If you set 'opterr' to
   zero, the error message is suppressed but we still return '?'.

   If a char in OPTSTRING is followed by a colon, that means it wants an
arg,
   so the following text in the same ARGV-element, or the text of the
following
   ARGV-element, is returned in 'optarg.  Two colons mean an option that
   wants an optional arg; if there is text in the current ARGV-element,
   it is returned in 'optarg'.

   If OPTSTRING starts with '-', it requests a different method of
handling the
   non-option ARGV-elements.  See the comments about
RETURN_IN_ORDER, above. */

int
GetOpt::operator () (void)
{
  if (nextchar == 0 || *nextchar == 0)
    {
      if (ordering == PERMUTE)
        {
          /* If we have just processed some options following some
non-options,
             exchange them so that the options come first. */
          if (first_nonopt != last_nonopt && last_nonopt != optind)
            exchange (nargv);
          else if (last_nonopt != optind)
            first_nonopt = optind;

          /* Now skip any additional non-options
             and extend the range of non-options previously skipped. */
          while (optind < nargc
                 && (nargv[optind][0] != '-'
                     || nargv[optind][1] == 0))
            optind++;
          last_nonopt = optind;
        }

      /* Special ARGV-element '--' means premature end of options.
         Skip it like a null option,
         then exchange with previous non-options as if it were an option,
         then skip everything else like a non-option. */
```

```
  if (optind != nargc && !strcmp (nargv[optind], "--"))
    {
      optind++;

      if (first_nonopt != last_nonopt && last_nonopt != optind)
        exchange (nargv);
      else if (first_nonopt == last_nonopt)
        first_nonopt = optind;
      last_nonopt = nargc;

      optind = nargc;
    }

  /* If we have done all the ARGV-elements, stop the scan
     and back over any non-options that we skipped and permuted. */
  if (optind == nargc)
    {
      /* Set the next-arg-index to point at the non-options
         that we previously skipped, so the caller will digest them. */
      if (first_nonopt != last_nonopt)
        optind = first_nonopt;
      return EOF;
    }

  /* If we have come to a non-option and did not permute it,
     either stop the scan or describe it to the caller and pass it by. */
  if (nargv[optind][0] != '-' || nargv[optind][1] == 0)
    {
      if (ordering == REQUIRE_ORDER)
        return EOF;
      optarg = nargv[optind++];
      return 0;
    }

  /* We have found another option-ARGV-element.
     Start decoding its characters. */

  nextchar = nargv[optind] + 1;
}

/* Look at and handle the next option-character. */

{
  char c = *nextchar++;
  char *temp = strchr (noptstring, c);

  /* Increment `optind' when we start to process its last character. */
  if (*nextchar == 0)
    optind++;

  if (temp == 0 || c == ':')
    {
      if (opterr != 0)
        {
          if (c < 040 || c >= 0177)
            fprintf (stderr, "%s:  unrecognized option, character
code 0%o\n",
                     nargv[0], c);
          else
            fprintf (stderr, "%s:  unrecognized option `-%c'\n",
                     nargv[0], c);
        }
      return '?';
    }
  if (temp[1] == ':')
    {
      if (temp[2] == ':')
        {
          /* This is an option that accepts an argument optionally. */
          if (*nextchar != 0)
            {
              optarg = nextchar;
              optind++;
            }
          else
            optarg = 0;
          nextchar = 0;
        }
      else
        {
          /* This is an option that requires an argument. */
          if (*nextchar != 0)
            {
              optarg = nextchar;
              /* If we end this ARGV-element by taking the rest as an
arg,
```

```
                we must advance to the next element now. */
              optind++;
            }
          else if (optind == nargc)
            {
              if (opterr != 0)
                fprintf (stderr, "%s:  no argument for '-%c'
option\n",
                         nargv[0], c);
              c = '?';
            }
          else
            /* We already incremented 'optind' once;
               increment it again when taking next ARGV-elt as
argument. */
            optarg = nargv[optind++];
          nextchar = 0;
        }
      }
    return c;
  }
}
// -*- C++ -*-
/* LineFlows.cpp - Jovan Ilic ilic@thetausa.com Sun Oct 29 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "Net.hpp"

bool Net::LineFlows() {
  int i, j, k, tb;
  double lineR, lineX, pGen, qGen;
  double ang, mag;
  complex<double> Y, eFB, eTB, eDiff, lineCurr, charging, chargingCurr;

  totalI2X = totalI2R = 0.0;
  totalCharging = 0.0;
  pGenTotal = qGenTotal = 0.0;
  pLoadTotal = qLoadTotal = 0.0;

  for(i = 0; i < gens.size(); i++) {

    k = gens[i].GenBusNo() - 1;
    pGenTotal += buses[k].PGen();
    qGenTotal += buses[k].QGen();
  }

  for(i = 0; i < buses.size(); i++) {

    pLoadTotal += buses[i].PLoad();
    qLoadTotal += buses[i].QLoad();

    ang = buses[i].V().imag();
    mag = buses[i].V().real();
    eFB = complex<double>(mag*cos(ang), mag*sin(ang));
    const vector<int> &tmpBus = buses[i].AdjBus();
    const vector<Branch *> &tmpBr = buses[i].AdjBranch();
    vector<complex<double> > &lf = buses[i].LineFlow();
    for(j = 0; j < tmpBus.size(); j++){
      tb = tmpBus[j] - 1;
      totalCharging += tmpBr[j]->BrBcap()*norm(eFB);

      Y = complex<double>(G.Aij(i,tb),B.Aij(i,tb));
      ang = buses[tb].V().imag();
      mag = buses[tb].V().real();
      eTB = complex<double>(mag*cos(ang), mag*sin(ang));
      eDiff = eFB - eTB;
      lineCurr = eDiff * Y;

      if(tb > i) {
lineR = -Y.real()/norm(Y);
lineX = Y.imag()/norm(Y);
totalI2R += lineR * norm(lineCurr);
totalI2X += lineX * norm(lineCurr);
      }

      charging = complex<double>(0.0, tmpBr[j]->BrBcap());
      chargingCurr = eFB * charging;
      lineCurr = chargingCurr - lineCurr;
      lineCurr = conj(lineCurr);
      lf[j] = lineCurr * eFB;
```

```
    }
  }

  return true;
}

// -*- C++ -*-
/* main.cpp - Jovan Ilic ilic@thetausa.com Fri Apr 21 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include <ga/ga.h>
#include <ga/GARealGenome.h>
#include <ga/GARealGenome.C>

#include "stds.hpp"
#include "GetOpt.h"

#define extern
#include "externs.hpp"
#undef extern

#include <imsl.h>

#include "Net.hpp"

Net tst;

extern float PowerCostGA(GAGenome &);
extern "C" void PowerCostTolmin(int, double *, double *);

int main(int argc, char **argv) {

  int i;

  InputFile_str = "none";
  OutGA_str = "none";
  OutTol_str = "none";
  SysOut_str = "none";
  GAParms_str = "none";

  interval = continuous = iZero = 0;

  vector<double> start;

  fstream outGA_fs, sysOut_fs, cfgGA_fs, errsGA_fs;
  fstream outTol_fs, inputFile_fs;

  GetOptions(argc, argv);

  if(InputFile_str == "none"){
    GetOptionsHelp(argc, argv);
    exit(-1);
  }else{
    inputFile_fs.open(InputFile_str.c_str(), ios::in);
    if(!inputFile_fs){
      cerr << "Could not open " << InputFile_str << " for reading" <<
endl;
      exit(-1);
    }
  }

  if(OutTol_str != "none") {
    outTol_fs.open(OutTol_str.c_str(), ios::out);
    if(!outTol_fs) {
      cerr << "Cannot open:  " << OutTol_str << " for reading" << endl;
      exit(-1);
    }
  }

  if(SysOut_str != "none") {
    sysOut_fs.open(SysOut_str.c_str(), ios::out);
    if(!sysOut_fs) {
      cerr << "Cannot open:  " << SysOut_str << " for writing" << endl;
      exit(-1);
    }
  }

  if(GAParms_str != "none") {
    cfgGA_fs.open(GAParms_str.c_str(), ios::in);
    if(!cfgGA_fs) {
```

```
      cerr << "Cannot open:  " << GAParms_str << " for reading" <<
endl;
      exit(-1);
    }
  }

  if(OutGA_str != "none") {
    outGA_fs.open(OutGA_str.c_str(), ios::out);
    if(!outGA_fs) {
      cerr << "Cannot open:  " << OutGA_str << " for writing" << endl;
      exit(-1);
    }
  }

  if(OutGA_str == "none" && OutTol_str == "none"){
    cerr << "At least one of -t or -g options is required" << endl;
    GetOptionsHelp(argc, argv);
    exit(-1);
  } else {
    tst.ReadAW(InputFile_str);
    tst.MakeY();
  }

  if(OutGA_str != "none") {
    int tmpOpts = interval + continuous + iZero;
    if(tmpOpts != 1){
      cerr << "One and only one of [-d|-c|-z] must be used." << endl;
      exit(-1);
    }

    vector<Generator> &gns = const_cast<vector<Generator>
&>(tst.GensV());
    int swng = tst.SwingBus();

    GARealAlleleSetArray PSysAlleles;

    if(interval) {
      for(i = 0; i < gns.size(); i++) {
  if(gns[i].GenBusNo() != tst.SwingBus()){
    PSysAlleles.add(gns[i].GenPMin()/tst.PBase(),
        gns[i].GenPMax()/tst.PBase(),
        (gns[i].GenPMax()/tst.PBase() -
gns[i].GenPMin()/tst.PBase())/10.0,
        GAAllele::INCLUSIVE,GAAllele::INCLUSIVE );
  }
      }
    }

    if(continuous) {
      for(i = 0; i < gns.size(); i++) {
  if(gns[i].GenBusNo() != tst.SwingBus()){
    PSysAlleles.add(gns[i].GenPMin()/tst.PBase(),
        gns[i].GenPMax()/tst.PBase(),
        GAAllele::INCLUSIVE,GAAllele::INCLUSIVE );
  }
      }

    }

    if(iZero) {
      float *allel;

      for(i = 0; i < gns.size(); i++) {
  if(gns[i].GenBusNo() != tst.SwingBus()){
    allel = new float[21];
    allel[0] = 0; cout << allel[0] << " ";
    for(int j = 1; j < 21; j++){
      allel[j] = gns[i].GenPMin()/100.0 + (j-1)*(gns[i].GenPMax() -
gns[i].GenPMin())/2000.0;
    }
    PSysAlleles.add(21, allel);
  }
      }
    }

    GARealGenome PSysGenome(PSysAlleles, PowerCostGA);

    // Now that we have the genomes, create a parameter list that will be
used for
    // all of the genetic algorithms and all of the genomes.

    GALinearScaling Scl;
    GAParameterList params;
    GASteadyStateGA::registerDefaultParameters(params);
    GASteadyStateGA PSysGa(PSysGenome);
```

```
    if(GAParms_str != "none") {
      params.read(GAParms_str.c_str());
    } else {
      params.set(gaNnGenerations, 50);
      params.set(gaNpopulationSize, 50);
      params.set(gaNscoreFrequency, 1);
      params.set(gaNflushFrequency, 10);
      params.set(gaNselectScores, GAStatistics::AllScores);
      params.set(gaNminimaxi, -1);
      PSysGa.set(gaNscoreFilename, "PSysGa.plt");
    }

    PSysGa.parameters(params);

    PSysGa.scaling(GANoScaling());

    PSysGa.terminator(GAGeneticAlgorithm::TerminateUponConvergence);

    PSysGa.evolve();

    // Save the "optimal" generation solution
    GARealGenome& g =
(GARealGenome&)PSysGa.statistics().bestIndividual();

    start.erase(start.begin(), start.end());

    int backIndx = 0;
    for(i = 0; i < gns.size(); i++) {
      if(gns[i].GenBusNo() == tst.SwingBus())
  backIndx = 1;
      else {
  // starting point for TOLMIN
  start.push_back(g.gene(i-backIndx));
  gns[i].GenP(g.gene(i-backIndx));
      }
    }

    tst.ResetBuses();
    tst.PFSolve();
    tst.LineFlows();

    cout << "Prod Cost = " << tst.ProdCost() << endl;

    for(i = 0; i < gns.size(); i++)
      cout << gns[i].CostInc(gns[i].GenP()*100.0) << '\t';

    cout << endl;

    if(outGA_fs.is_open()){
      tst.PrintAWOut(outGA_fs);
      outGA_fs.close();
    }

    if(errsGA_fs.is_open())
      errsGA_fs.close();
  }

  // Calculate the same optimal point using
  // quadratic programming
  if(OutTol_str != "none") {
    vector<Generator> &gns = const_cast<vector<Generator>
&>(tst.GensV());
    int swng = tst.SwingBus();

    // number of equations
    int neq = 0;

    // equations
    double a[ ] = {0};
    double b[ ] = {0};

    // number of variables
    int nvar = gns.size() - 1;

    cerr << "nvar = " << nvar << endl;

    // number of constraints
    // there are only simple bounds
    int ncon = 0;

    double *xlb = new double[nvar];
    double *xub = new double[nvar];

    int swngFound = 0, i;
```

```
  for(i = 0; i < gns.size(); i++) {
    if(gns[i].GenBusNo() != tst.SwingBus()){
xlb[i - swngFound] = gns[i].GenPMin()/tst.PBase();
xub[i - swngFound] = gns[i].GenPMax()/tst.PBase();
    } else {
swngFound = 1; // offset for the swing bus
    }
  }

  double *x = new double [nvar];

  if(OutGA_str != "none"){
    imsl_d_write_matrix("Solution start ", 1, nvar, &start[0], 0);

    imsl_d_min_con_gen_lin(PowerCostTolmin, nvar, ncon,
          neq, a, b, xlb, xub, IMSL_RETURN_USER,
          x, IMSL_MAX_FCN, 10000,
          IMSL_XGUESS, &start[0], 0);
  } else {
    imsl_d_min_con_gen_lin(PowerCostTolmin, nvar, ncon,
          neq, a, b, xlb, xub, IMSL_RETURN_USER,
          x, IMSL_MAX_FCN, 10000, 0);
  }

  imsl_d_write_matrix("Solution", 1, nvar, x, 0);
  swngFound = 0;

  for(i = 0; i < nvar; i++) {
    if(gns[i].GenBusNo() != tst.SwingBus()) {
gns[i].GenP(x[i - swngFound]);
    }else{
swngFound = 1;
    }
  }

  tst.ResetBuses();
  tst.PFSolve();
  tst.LineFlows();

  cout << "Prod Cost = " << tst.ProdCost() << endl;

  for(i = 0; i < gns.size(); i++)
    cout << gns[i].CostInc(gns[i].GenP()*100.0) << '\t';

  cout << endl;

  if(outTol_fs.is_open()){
    tst.PrintAWOut(outTol_fs);
    outTol_fs.close();
  }

  delete [ ] xlb;
  delete [ ] xub;

 }

 if(sysOut_fs.is_open()){
   tst.PrintAW(sysOut_fs);
   sysOut_fs.close();
 }

 return 0;
}
// -*- C++ -*-
/* MakeGB.cpp - Jovan Ilic ilic@thetausa.com Sun Oct 15 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "Net.hpp"


// bool Net::MakeGB() calculates the admittance matrix
// components G and B. Parallel lines and transformers
// complicate the situation a little bit.
bool Net::MakeY() {
  int i, j, k, gColIndx, bColIndx;
  double x, r, bcap, diagG, diagB, ysh;
  vector<int> doneBus;
  // alocate memory for column indecies
  xg.erase(xg.begin(), xg.end());
  xb.erase(xb.begin(), xb.end());
```

```
xg.reserve(noBus + 1);
xb.reserve(noBus + 1);
// alocate memory for row indecies and data
g.erase(g.begin(), g.end());
b.erase(b.begin(), b.end());
gsub.erase(gsub.begin(), gsub.end());
bsub.erase(bsub.begin(), bsub.end());
j = 0;
for(i = 0; i < buses.size(); i++){
  vector<int> tmpBus = buses[i].AdjBus();
  for(k = 0; k < tmpBus.size(); k++)
    if(tmpBus[k] < i+1)
j++;
}
j += buses.size();
g.reserve(j);
b.reserve(j);
gsub.reserve(j);
bsub.reserve(j);
gColIndx = bColIndx = 0;
xg.push_back(gColIndx);
xb.push_back(bColIndx);

for(i = 0; i < buses.size(); i++) {
  diagG = diagB = 0.0;
  const vector<Branch *> &tmpBr = buses[i].AdjBranch();
  const vector<int> &tmpBus = buses[i].AdjBus();
  doneBus.erase(doneBus.begin(), doneBus.end());
  // first calculate the diagonal element
  for(j = 0; j < tmpBr.size(); j++) {
    x = tmpBr[j]→BrX();
    r = tmpBr[j]→BrR();
    bcap = tmpBr[j]→BrBcap();
    double z = r*r + x*x;
    double gline = r/z;
    double bline = -x/z;
    ysh = buses[i].YSh();
    // TRAN line has no bcap!
    if(tmpBr[j]→BrLineType() == "TRAN") {
double a = tmpBr[j]→BrTapRatio();
diagG += gline/a;
diagB += bline/a + bcap;
if(tmpBr[j]→BrTapBus() == i){
  diagG += (1.0 - a)/(a*a) * gline;
  diagB += (1.0 - a)/(a*a) * bline;
} else {
  diagG += (a - 1.0)/(a) * gline;
  diagB += (a - 1.0)/(a) * bline;
}
    } else {
diagG += gline;
diagB += bline + bcap;
    }
  }
  // store the diagonal element
  // diagonal elements are always nonzero
  // but it should still be checked
  if(diagB != 0.0) {
    bsub.push_back(i);
    b.push_back(diagB + ysh);
    bColIndx++;
  }
  if(diagG != 0) {
    g.push_back(diagG);
    gsub.push_back(i);
    gColIndx++;
  }
  // calcualte and store elemets below the diagonal
  for(j = 0; j < tmpBr.size(); j++) {
    double z = 0.0;
    double gline = 0.0;
    double bline = 0.0;
    if(tmpBus[j] < i + 1)
  continue; // don't count branches twice
    if(find(doneBus.begin(), doneBus.end(), tmpBus[j]) !=
doneBus.end())
  continue; // already accounted for parallel line
    doneBus.push_back(tmpBus[j]);
    for(int k = j; k < tmpBus.size(); k++) {
  if(tmpBus[j] == tmpBus[k]) { // sum up all the parallel lines
    x = tmpBr[k]→BrX();
    r = tmpBr[k]→BrR();
    bcap = tmpBr[k]→BrBcap();
    z = r*r + x*x;
    if(tmpBr[k]→BrLineType() == "TRAN"){
      double a = tmpBr[k]→BrTapRatio();
```

```
        gline += (r/z)/a;
        bline += (-x/z)/a;
      } else {
        gline += r/z;
        bline += -x/z;
      }
    }
        }
        // bline or gline can be zero but not both at the same time
        if(bline != 0.0) {
    bsub.push_back(tmpBus[j] - 1);
    b.push_back(-bline);
    bColIndx++;
        }
        if(gline != 0.0) {
    gsub.push_back(tmpBus[j] - 1);
    g.push_back(-gline);
    gColIndx++;
        }
      }
      // indecies increased by at least one
      xg.push_back(gColIndx);
      xb.push_back(bColIndx);
    }
    G.SetA(g, gsub, xg, SYL);
    B.SetA(b, bsub, xb, SYL);
    return true;
}



// -*- C++ -*-
/* MakeJac.cpp - Jovan Ilic ilic@thetausa.com Sun Oct 15 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "Net.hpp"

// memory requirements for different vectors need to be
// recalculated.
bool Net::MakeJ() {
  int i, j, k;
  // counts of PQ, PV, and PC buses.
  PQCount=0, PVCount=0, VCCount=0;
  for(i = 0; i < buses.size(); i++){
    switch (buses[i].Type()) {
    case LOAD :
    case GEN_HIGHVAR :
    case GEN_LOWVAR :
      PQCount++;
      break;
    case GEN_REG :
    case GEN :
      PVCount++;
      break;
    default :
      ; // probably a SWING bus
      break;
    }
  }
  jac.erase(jac.begin(), jac.end());
  jacsub.erase(jacsub.begin(), jacsub.end());
  xjac.erase(xjac.begin(), xjac.end());
  // dP/dT and dP/dE rows might be shifted up by 1 if row index is
  // greater than the swingBus. dQ/dT and dQ/dE shift will depend on
  // the number of generator buses encountered before current row index.
  rowShifts.erase(rowShifts.begin(), rowShifts.end());
  rowShifts.reserve(buses.size());
  for(i = 0, j = 0; i < buses.size(); i++) {
    if(buses[i].Type() == GEN || buses[i].Type() == SWING)
      j++;
    rowShifts.push_back(j);
  }
  delPQ.erase(delPQ.begin(), delPQ.end());
  delPQ.reserve(buses.size() - 1 + (buses.size() - 1 - PVCount));
  for(i = 0; i < (buses.size() - 1 + (buses.size() - 1 - PVCount)); i++)
    delPQ.push_back(0.0);
  // memory requirements for the Jacobian could
  // be estimated here as well
```

```
  DelPQ(); // Calculates NetInj() as well
  int elCount = 0;
  int rowShift = 0;
  double der;
  xjac.push_back(elCount);
  int tb, fb;
  // calculate dP/dT part first
  for(k = 0; k < buses.size(); k++) {
    if(buses[k].Type() == SWING)
      continue;
    tb = k;
    // calculate the diagonal term
    der = -buses[tb].QNetInj() -
B.Aij(tb,tb)*buses[tb].V().real()*buses[tb].V().real();
    rowShift = tb > (swingBus - 1) ? 1 : 0;
    jacsub.push_back(tb - rowShift);
    jac.push_back(der);
    elCount++;
    // get a list of unique buses connected to bis i.
    const vector<int> &tmpFB = buses[tb].AdjBus();
    // dP/dT
    for(i = 0; i < tmpFB.size(); i++) {
      if(find(tmpFB.begin(), &tmpFB[i], tmpFB[i]) != &tmpFB[i])
  continue; // this branch already accounted for
      fb = tmpFB[i] - 1;
      if(buses[fb].Type() == SWING)
  continue;
      rowShift = fb > (swingBus - 1) ? 1 : 0;
      jacsub.push_back(fb - rowShift);
      elCount++;
      der = buses[tb].V().real()*buses[fb].V().real()* \
  (G.Aij(tb,fb)*sin(buses[tb].V().imag() - buses[fb].V().imag()) - \
   B.Aij(tb,fb)*cos(buses[tb].V().imag() - buses[fb].V().imag()));
      jac.push_back(der);
    }
    // first, calculate the diagonal element
    if((buses[tb].Type() != SWING) && (buses[tb].Type() != GEN)) {
      der = buses[tb].PNetInj() -
G.Aij(tb,tb)*buses[tb].V().real()*buses[tb].V().real();
      jac.push_back(der);
      jacsub.push_back(buses.size() - 1 + tb - rowShifts[tb]);
      elCount++;
    }
    // dQ/dT
    for(i = 0; i < tmpFB.size(); i++) {
      if(find(tmpFB.begin(), &tmpFB[i], tmpFB[i]) != &tmpFB[i])
  continue; // this branch already accounted for
      fb = tmpFB[i] - 1;
      rowShift = fb > (swingBus - 1) ? 1 : 0;
      if(buses[fb].Type() == SWING || buses[fb].Type() == GEN)
  continue;
      der = -buses[tb].V().real()*buses[fb].V().real()* \
  (G.Aij(tb,fb)*cos(buses[tb].V().imag() - buses[fb].V().imag()) - \
   B.Aij(tb,fb)*sin(buses[i].V().imag() - buses[fb].V().imag()));
      jac.push_back(der);
      jacsub.push_back(fb + buses.size() - 1 - rowShifts[fb]);
      elCount++;
    }
    xjac.push_back(elCount);
  }
  for(k = 0; k < buses.size(); k++) {
    if((buses[k].Type() == GEN) || (buses[k].Type() == SWING))
      continue;
    tb = k;
    vector<int> tmpFB = buses[k].AdjBus();
    // calculate the diagonal term
    der = buses[tb].PNetInj() +
G.Aij(tb,tb)*buses[tb].V().real()*buses[tb].V().real();
    jac.push_back(der);
    rowShift = tb > (swingBus - 1) ? 1 : 0;
    jacsub.push_back(tb - rowShift);
    elCount++;
    // dP/d|E|/|E|
    for(i = 0; i < tmpFB.size(); i++) {
      if(find(tmpFB.begin(), &tmpFB[i], tmpFB[i]) != &tmpFB[i])
  continue; // this branch already accounted for
      fb = tmpFB[i] - 1;
      if(buses[fb].Type() == SWING )
        continue;
      der = buses[tb].V().real()*buses[fb].V().real()* \
  (G.Aij(tb,fb)*cos(buses[tb].V().imag() - buses[fb].V().imag()) - \
   B.Aij(tb,fb)*sin(buses[tb].V().imag() - buses[fb].V().imag()));
      jac.push_back(der);
      rowShift = fb > (swingBus - 1) ? 1 : 0;
      jacsub.push_back(fb - rowShift);
      elCount++;
```

```
    }
    // dQ/d|E|/|E|
    der = buses[tb].QNetInj() -
B.Aij(tb,tb)*buses[tb].V().real()*buses[tb].V().real();
    jac.push_back(der);
    jacsub.push_back(buses.size() - 1 + tb - rowShifts[tb]);
    elCount++;
    for(i = 0; i < tmpFB.size(); i++) {
      if(find(tmpFB.begin(), &tmpFB[i], tmpFB[i]) != &tmpFB[i])
  continue; // this branch already accounted for
      fb = tmpFB[i] - 1;
      if(buses[fb].Type() == SWING || buses[fb].Type() == GEN)
        continue;
      der = buses[tb].V().real()*buses[fb].V().real()* \
  (G.Aij(tb,fb)*sin(buses[tb].V().imag() - buses[fb].V().imag()) - \
   B.Aij(tb,fb)*cos(buses[tb].V().imag() - buses[fb].V().imag()));
      jac.push_back(der);
      jacsub.push_back(fb + buses.size() - 1 - rowShifts[fb]);
      elCount++;
    }
    xjac.push_back(elCount);
  }
  // J.SetA(xjac.size()-1,xjac.size()-1, jac.size(), jac, jacsub, xjac, GE);
  J.SetA(jac, jacsub, xjac, GE); // don't remember why -1
  return true;
}

bool Net::NetInj() {
  int i, j, k;
  double p, q;
  // Calculate pNetInj and qnetInj
  for(i = 0; i < buses.size(); i++) {
    p = buses[i].V().real()*buses[i].V().real() * G.Aij(i,i);
    q = -buses[i].V().real()*buses[i].V().real() * B.Aij(i,i);
    const vector<int> &tmpBus = buses[i].AdjBus();
    for( j = 0; j < tmpBus.size(); j++) {
      if(find(tmpBus.begin(), &tmpBus[j], tmpBus[j]) != &tmpBus[j])
  continue; // this branch already accounted for
      k = tmpBus[j] - 1;
      p += buses[i].V().real()*buses[k].V().real() * \
  (G.Aij(i,k)*cos(buses[i].V().imag() - buses[k].V().imag()) + \
   B.Aij(i,k)*sin(buses[i].V().imag() - buses[k].V().imag()));
      q += buses[i].V().real()*buses[k].V().real()* \
  (G.Aij(i,k)*sin(buses[i].V().imag() - buses[k].V().imag()) - \
   B.Aij(i,k)*cos(buses[i].V().imag() - buses[k].V().imag()));
    }
    buses[i].PNetInj(p);
    buses[i].QNetInj(q);
  }

  // tell swing bus whats its generation
  buses[swingBus - 1].Gen()→GenP(buses[swingBus - 1].PLoad() +
buses[swingBus - 1].PNetInj());
  return true;

}

// should be called after MakeJ() is called
bool Net::DelPQ() {
  int i, j;
  NetInj();
  int rowIndx = 0;
  for(i = 0; i < buses.size(); i++) {
    if(buses[i].Type() == SWING)
      continue;
    delPQ[rowIndx++] = buses[i].PSched() - buses[i].PNetInj();
  }
  for(i = 0; i < buses.size(); i++){
    if(buses[i].Type() == GEN || buses[i].Type() == SWING)
      continue;
    delPQ[rowIndx++] = buses[i].QSched() - buses[i].QNetInj();
  }
  pMismatch = qMismatch = 0.0;
  rowIndx = 0;
  for(i = 0; i < buses.size(); i++){
    if(buses[i].Type() == SWING)
      continue;
    pMismatch = fabs(delPQ[rowIndx]) > pMismatch ?
fabs(delPQ[rowIndx]) : pMismatch;
    qMismatch = fabs(delPQ[rowIndx]) > qMismatch ?
fabs(delPQ[rowIndx]) : qMismatch;
    rowIndx++;
  }
  for(i = 0; i < buses.size(); i++){
    if(buses[i].Type() == SWING || buses[i].Type() == GEN)
      continue;
```

```
    pMismatch = fabs(delPQ[rowIndx]) > pMismatch ?
fabs(delPQ[rowIndx]) : pMismatch;
    qMismatch = fabs(delPQ[rowIndx]) > qMismatch ?
fabs(delPQ[rowIndx]) : qMismatch;
    rowIndx++;
  }
  return true;
}

// -*- C++ -*-
/* MatrixSPD.cpp - Jovan Ilic ilic@thetausa.com Fri Dec 29 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "MatrixSPD.hpp"

MatrixSPD::MatrixSPD() {
  atype = GE;
  m = 0;
  n = 0;
  nnz = 0;
  HBvalid = false;
}

MatrixSPD::MatrixSPD(int i, int j) {
  int it, jt;
  atype = GE;
  m = i;
  n = j;
  nnz = 0;
  xa.erase(xa.begin(), xa.end());
  asub.erase(asub.begin(), asub.end());
  a.erase(a.begin(), a.end());
  xa.reserve(j);
  dat.erase(dat.begin(), dat.end());
  for(it = 0; it < j; it++)
    dat.push_back(map<int, double, less<int> >());
  HBvalid = false;
}

MatrixSPD::MatrixSPD(int i, int j, int nz) {
  int it, jt;
  atype = GE;
  m = i;
  n = j;
  nnz = nz;
  // initialize and reserve memory for HB arrays
  xa.erase(xa.begin(), xa.end());
  asub.erase(asub.begin(), asub.end());
  a.erase(a.begin(), a.end());
  xa.reserve(j+1);
  asub.reserve(nz);
  a.reserve(nz);
  // initialize (not set) data map
  dat.erase(dat.begin(), dat.end());
  for(it = 0; it < j; it++)
    dat.push_back(map<int, double, less<int> >());
  HBvalid = false;
}

MatrixSPD::MatrixSPD(const MatrixSPD &mspd) {
  if(this == &mspd)
    return;
  atype = mspd.Atype();
  m = mspd.M();
  n = mspd.N();
  nnz = mspd.NNZ();
  a = mspd.A();
  asub = mspd.Asub();
  xa = mspd.Xa();
  dat.erase(dat.begin(), dat.end());
  dat.reserve(n);
  for(int i = 0; i < n; i++)
    dat.push_back(mspd.Col(i));
  HBvalid = mspd.HBValid();
}


MatrixSPD &MatrixSPD::operator=(const MatrixSPD &mspd) {
  if(this == &mspd)
    return *this;
```

```
  atype = mspd.Atype();
  m = mspd.M();
  n = mspd.N();
  nnz = mspd.NNZ();
  a = mspd.A();
  asub = mspd.Asub();
  xa = mspd.Xa();
  dat.erase(dat.begin(), dat.end());
  dat.reserve(n);
  for(int i = 0; i < n; i++)
    dat.push_back(mspd.Col(i));
  HBvalid = mspd.HBValid();
  return *this;
}

bool MatrixSPD::SetA(vector<double> &d, vector<int> &rowp, \
        vector<int> &colp, Mtype_t tp, int rc){
  int i;
  // here we let STL manage vectors
  atype = tp;
  // number of rows is not really known here.
  n = colp.size() - 1;
  if(rc != 0)
    m = rc;
  else
    m = n;
  nnz = d.size();
  a = d;
  asub = rowp;
  xa = colp;
  // set the maps
  dat.erase(dat.begin(), dat.end());
  dat.reserve(n);
  for(i = 0; i < n; i++)
    dat.push_back(map<int, double, less<int> >());
  int rowCnt = 0;
  for(i = 1; i < xa.size(); i++) {
    for(int j = 0; j < xa[i] - xa[i-1]; j++){
      dat[i-1][asub[rowCnt]] = a[rowCnt];
      rowCnt++;
    }
  }
  HBvalid = true;
  return true;
}

bool MatrixSPD::Solve(vector<double> &b, vector<double> &x) {
  Imsl_d_sparse_elem aa[ ]= {0,0,0};
  if(b.size() > x.size())
    return false;
  UpdateHB();
  imsl_d_lin_sol_gen_coordinate(n, nnz, aa, &b[0], IMSL_CSC_FORMAT,
            &xa[0], &asub[0], &a[0],
            IMSL_RETURN_USER, &x[0], 0);
  return true;
}


// this is a tricky part that needs a better solution.
double MatrixSPD::Aij(const int i, const int j, const double d) {
  if(j > n || i > m || d == 0){
    return 0.0;
  }
  if(dat[j].find(i) == dat[j].end())
    dat[j][i] = d;
  else
    dat[j].find(i)->second = d;
  HBvalid = false;
  return d;

}


double MatrixSPD::Aij(int ii, int jj) const {
  int tmp, i = ii, j = jj;
  if((atype == SYU) && (i > j)){
    tmp = i;
    i = j;
    j = tmp;
  }
  if((atype == SYL) && (i < j)){
    tmp = i;
    i = j;
    j = tmp;
  }
  if(j > n || i > m){
```

```
    return 0.0;
  }
  if(dat[j].find(i) == dat[j].end() ){
    return 0.0;
  }else{
    return(dat[j].find(i)→second);
  }
}

// access HB column maps
const map<int, double, less<int> > &MatrixSPD::Col(int jc) const {
return dat[jc];}
void MatrixSPD::Col(const map<int, double, less<int> > &jm, int ji) {
  if(ji < m){
    dat[ji] = jm;
    HBvalid = false;
  }
}

bool MatrixSPD::HBValid() const { return HBvalid; }
void MatrixSPD::HBValid(const bool hb) { HBvalid = hb;}

void MatrixSPD::PrintMap(ostream &os) {
  int i, j;
  map<int, double, less<int> >::iterator mp;
  for(i = 0; i < dat.size(); i++){
    for(mp = dat[i].begin(); mp != dat[i].end(); mp++)
      os ≪ '(' ≪ i ≪ ',' ≪ (*mp).first ≪ ") = " ≪ (*mp).second ≪ endl;
  }
}

void MatrixSPD::PrintHB(ostream &os) {
  int i, j, k;
  UpdateHB();
  for(i = 0; i < xa.size(); i++)
    os ≪ xa[i] ≪ " ";
  os ≪ endl;
  for(i = 0; i < asub.size(); i++)
    os ≪ asub[i] ≪ " ";
  os ≪ endl;
  for(i = 0; i < a.size(); i++)
    os ≪ a[i] ≪ " ";
  os ≪ endl;
}

// not very efficient but since it prints out a dense format
// of a mtrix, it should not be used with large matrices anyway
ostream &operator≪(ostream &os, const MatrixSPD &msp) {
  int i, j;
  for(i = 0; i < msp.M(); i++){
    for(j = 0; j < msp.N(); j++)
      os ≪ msp.Aij(i,j) ≪ '\t';
    os ≪ endl;
  }
  return os;
}

void MatrixSPD::UpdateHB() {
  if(HBvalid)
    return;
  int i,j,k;
  map<int, double, less<int> >::iterator dp;
  a.erase(a.begin(), a.end());
  xa.erase(xa.begin(), xa.end());
  asub.erase(asub.begin(), asub.end());
  xa.reserve(n + 1);
  xa.push_back(0);
  j = 0;
  for(i = 0; i < dat.size(); i++){
    k = dat[i].size();
    xa.push_back(k);
    j += k;
  }
  a.reserve(j); asub.reserve(j);
  for(i = 0; i < dat.size(); i++) {
    for(dp = dat[i].begin(); dp != dat[i].end(); dp++){
      asub.push_back(dp→first);
      a.push_back(dp→second);
    }
  }
}

// -*- C++ -*-
/* Net.cpp - Jovan Ilic ilic@thetausa.com Tue Sep 5 2000
 *
 * $Log$
```

```
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "Net.hpp"

extern bool lessTB(const Branch &, const Branch &);
extern bool lessFB(const Branch &, const Branch &);
extern bool lessGen(const Generator &, const Generator &);
extern bool lessBus(const Bus &, const Bus &);

bool Net::ReadAW(const string &f) {
  char tmpLn[512];
  int i, tmpI;
  double tmpD;
  string tmpStr;
  fstream fs(f.c_str(), ios::in);
  if(!fs) {
    cerr << "Could not open:  " << f << " for reading." << endl;
    exit(-1);
  }
  // Get two title lines
  fs.getline(tmpLn, 512); Title1 = tmpLn;
  fs.getline(tmpLn, 512); Title2 = tmpLn;
  // read number of buses, number of generators
  // number of branches, and swing bus number
  fs >> noBus >> noGen >> noBr >> swingBus >> noCap;
  // read bus data
  for(i = 0; i < noBus; i++) {
    buses.push_back(Bus());
    fs >> tmpI; buses[i].BusNo(tmpI);
    fs >> tmpStr; buses[i].BusName(tmpStr);
    fs >> tmpD; buses[i].PLoad(tmpD);
    fs >> tmpD; buses[i].QLoad(tmpD);
    fs >> tmpD; buses[i].VLL(tmpD);
    fs >> tmpD; buses[i].VHL(tmpD);
    buses[i].BusType("LOAD"); // default
    buses[i].Type(LOAD); // default
  }
  sort(buses.begin(), buses.end(), lessBus);
  // read in generator data
  for(i = 0; i < noGen; i++) {
    gens.push_back(Generator());
    fs >> tmpI; gens[i].GenBusNo(tmpI);
    buses[tmpI-1].BusType("GEN"); buses[tmpI-1].Type(GEN);
    fs >> tmpD; gens[i].GenP(tmpD);
    fs >> tmpD; gens[i].GenV(tmpD);
    fs >> tmpD; gens[i].GenQMax(tmpD);
    fs >> tmpD; gens[i].GenQMin(tmpD);
    fs >> tmpD; gens[i].GenPMin(tmpD);
    fs >> tmpD; gens[i].GenPMax(tmpD);
    // get the rest of the line which contains cost
    // coefficients in increasing order
    fs.getline(tmpLn, 512);
    istrstream tmpStr(tmpLn);
    while(tmpStr){
      tmpStr >> tmpD;
      if(tmpStr)
gens[i].CostPoly(tmpD);
    }
  }
  buses[swingBus-1].BusType("SWING");
  buses[swingBus-1].Type(SWING);
  sort(gens.begin(), gens.end(), lessGen);
  // this must be done after sorting gens;
  for(i = 0; i < gens.size(); i++){
    int busIndx;
    busIndx = gens[i].GenBusNo() - 1;
    buses[busIndx].Gen(&gens[i]); // attach generator to bus
    buses[busIndx].V(gens[i].GenV());
  }
  // read in branches data
  noTran = 0;
  trans.erase(trans.begin(), trans.end());
  for(i = 0; i < noBr; i++) {
    branches.push_back(Branch());
    fs >> tmpI; branches[i].BrFB(tmpI);
    fs >> tmpI; branches[i].BrTB(tmpI);
    fs >> tmpD; branches[i].BrR(tmpD);
    fs >> tmpD; branches[i].BrX(tmpD);
    fs >> tmpD; branches[i].BrBcap(tmpD);
    fs >> tmpD; branches[i].BrLineFlowLimit(tmpD);
    fs >> tmpLn; branches[i].BrLineType(tmpLn);
    if(branches[i].BrLineType() == "TRAN"){
```

```
      noTran++;
      trans.push_back(tmpI);
      fs >> tmpI; branches[i].BrTapBus(tmpI);
      fs >> tmpI; branches[i].BrNonTapBus(tmpI);
      fs >> tmpD; branches[i].BrTapRatio(tmpD);
    }
  }
  sort(trans.begin(), trans.end());
  // order branches according to "from node" first ....
  sort(branches.begin(), branches.end(), lessFB);
  vector<Branch>::iterator cur, left;
  left = branches.begin();
  // then according to "to node".
  for(cur = branches.begin(); cur < branches.end(); cur++){
    if((*left).BrFB() != (*cur).BrFB()){
      sort(left, cur, lessTB);
      left = cur;
    }
  }
  // Get the capacitors in the bus data;
  caps.erase(caps.begin(), caps.end());
  caps.reserve(noCap);
  for(i = 0; i < noCap; i++){
    fs >> tmpI >> tmpD;
    caps.push_back(tmpI);
    buses[tmpI - 1].YSh(tmpD);
  }
  sort(caps.begin(), caps.end());
  // this is good place to finalize Bus::adjBranch
  // Bus::adjBranch shold be erased here first
  // The destination bus read from a Branch is either
  // Branch::BrFB() or Branch::BrTB() depending on which
  // which one is the current bus (== "bus number" + 1)
  // This is probably an issue with tap transformers only.
  for(i = 0; i < branches.size(); i++) {
    buses[branches[i].BrFB()-1].AdjBranch(&branches[i]);
    buses[branches[i].BrTB()-1].AdjBranch(&branches[i]);

  }
  // Tracing the adjacent buses should make programming really
  // easier and future exapansion simpler.
  for(i = 0; i < buses.size(); i++){
    vector<Branch *> tmpBr = buses[i].AdjBranch();
      for(int j = 0; j < tmpBr.size(); j++){
        if(tmpBr[j]->BrFB() == i+1)
      buses[i].AdjBus(tmpBr[j]->BrTB());
        else
      buses[i].AdjBus(tmpBr[j]->BrFB());
        buses[i].LineFlow().push_back(0);
      }
    }
    return true;
  }

  bool Net::ResetBuses() {

    for(int i = 0; i < buses.size(); i++) {
      if(buses[i].Type() != SWING)
        if(buses[i].Gen() != NULL){
    buses[i].V(buses[i].Gen()->GenV());
    buses[i].QVarLimit(0.0);
    buses[i].Type(GEN);
    buses[i].BusType("GEN");
        } else {
    buses[i].V(1.0);
    buses[i].QVarLimit(0.0);
    buses[i].Type(LOAD);
    buses[i].BusType("LOAD");
        }
    }
    PMismatch(2.0*PMaxMismatch());
    QMismatch(2.0*QMaxMismatch());
    return true;
  }

  bool Net::PrintAW(ostream &os) {
    int i;
    ios::fmtflags oldFlags = os.flags();
    os << endl << endl << "Input File Name:  " << InputFile_str << endl <<
  endl << endl;
    os << Title1 << endl;
    os << Title2 << endl;
    os << " Number of Buses = " << "\t\t" << noBus << endl;
    os << " Number of Lines = " << "\t\t" << noBr << endl;
    os << " Number of Generators = " << "\t" << noGen << endl;
    os << " Number of Transformers = " << "\t" << noTran << endl;
```

```
  os << " Swing Bus at bus number " << "\t" << swingBus << endl;
  os << endl << "Bus Data" << endl << endl;
  os << "Bus Voltage Voltage Bus " << endl;
  os << "Number Bus Name Mw Load Mvar Load High Limit Low limit
Type " << endl;
  os << "------ -------- ---------- ---------- ----------
---------- -----" << endl;
  for(i = 0; i < noBus; i++)
    os << buses[i];
  os << endl;
  os << "Generator Data" << endl << endl;
  os << "Bus Mw Sched H Mvar L MVar Low P High P C B A " << endl;
  os << "number Gen.  Volt.  Limit Limit Limit Limit " << endl;
  os << "------ ----- ------ ------- ------- ------ ------- -------
------- ------- -------" << endl;
  for(i = 0; i < noGen; i++)
    os << gens[i];
  os << endl;
  os << "Branch Data" << endl;
  os << endl;
  os << "From To Branch Branch Line Branch Branch Tap Non-Tap Tap "
<< endl;
  os << "Bus Bus R X Charging Mva Limit Type Bus Bus Ratio " << endl;
  os << "---- ---- ------ ------ -------- --------- ------ ----
------- ------" << endl;
  for(i = 0; i < noBr; i++)
    os << branches[i];
  os.setf(ios::left, ios::adjustfield);
  // List of capacitors with bus indecies here.
  os << endl << "Capacitors List" << endl << endl;
  os << "Bus Number " << "Capacitor Value" << endl;
  os << "---------- " << "---------------" << endl;
  for(i = 0; i < noCap; i++) {
    os << setw(10) << setprecision(5) << caps[i] << " " <<
buses[caps[i]-1].YSh() << endl;
  }
  os.flags(oldFlags);
  return true;
}

bool Net::PrintAWOut(ostream &os) {
  int i,j,k,fb,tb;
  double r2d = 180.0/M_PI;
  ios::fmtflags oldFlags = os.flags();
  os.setf(ios::right, ios::adjustfield);
  os.setf(ios::fixed, ios::floatfield);
  os <<
"=========================================";
  os <<
"=========================================="
<< endl;
  os << "Solution Output" << endl;
  os <<
"=========================================";
  os <<
"=========================================="
<< endl;
  os << endl;
  os << "Load Flow Solution:" << endl;
  os << endl;
  os << "file:  " << InputFile_str << endl;
  os << endl;
  os << Title1 << endl;
  os << Title2 << endl;
  os << endl;
  os << "From Volt.  Volt.  Mw Mvar Mw Mvar To Mw Mvar Mva " << endl;
  os << "Bus Mag.  Angle Load Load Gen.  Gen.  Bus Flow Flow Flow "
<< endl;
  os << "---- ----- ------- ------- ------- ------- ------- ----
------- ------- -------" << endl;
  os << endl;
  for(i = 0; i < buses.size(); i++) {
    os << endl;
    os << setw(4) << (i + 1) << " ";
    os << setw(5) << setprecision(3) << buses[i].V().real() << " ";
    os << setw(7) << setprecision(3) << (r2d*buses[i].V().imag()) << " ";
    os << setw(7) << setprecision(2) << (pBase*buses[i].PLoad()) << " ";
    os << setw(7) << setprecision(2) << (pBase*buses[i].QLoad()) << " ";
    if(buses[i].Gen() != NULL){
      os << setw(7) << setprecision(2) << (pBase*buses[i].PGen()) << " ";
      os << setw(7) << setprecision(2) << (pBase*buses[i].QGen()) << " ";
    } else {
      os << setw(7) << setprecision(2) << 0.0 << " ";
      os << setw(7) << setprecision(2) << 0.0 << " ";
    }
    os << endl;
```

```
    const vector<int> &tmpBus = buses[i].AdjBus();
    const vector<complex<double> > &lf = buses[i].LineFlow();
    for(j = 0; j < tmpBus.size(); j++) {
      for (k = 1; k <= 51; k++)
  os << ' ';
      os << setw(4) << tmpBus[j] << " ";
      os << setw(7) << setprecision(2) << (pBase*lf[j].real()) << " ";
      os << setw(7) << setprecision(2) << (pBase*lf[j].imag()) << " ";
      os << setw(7) << setprecision(2) << (pBase*abs(lf[j])) << endl;
    }
  }
  os << endl << endl;
  os << "Mw gen = " << setw(7) << setprecision(3) << (pBase*pGenTotal)
<< endl;
  os << "Mvar gen = " << setw(7) << setprecision(3) << (pBase*qGenTotal)
<< endl;
  os << "Mw load = " << setw(7) << setprecision(3) << (pBase*pLoadTotal)
<< endl;
  os << "Mvar load = " << setw(7) << setprecision(3) <<
(pBase*qLoadTotal) << endl;
  os << "Total I2R Mw losses = " << setw(7) << setprecision(3) <<
(pBase*totalI2R) << endl;
  os << "Total I2X Mvar losses = " << setw(7) << setprecision(3) <<
(pBase*totalI2X) << endl;
  os << "Total charging Mvar = " << setw(7) << setprecision(3) <<
(pBase*totalCharging) << endl;
  os <<
"=========================================";
  os <<
"=========================================="
<< endl;
  os << " End of Output " << endl;
  os <<
"=========================================";
  os <<
"=========================================="
<< endl;
  os.flags(oldFlags);
  return true;
}
/* - J.Ilic e:/ilic/nn/pnn/specht/options.cc
 * Wed Oct 14 1992
 * <ilic@morse.eece.maine.edu>
 */

// $Log$

#ifndef lint
static char *rcsid = "$Id";
#endif

#include "stds.hpp"

#include "GetOpt.h"

#include "externs.hpp"


// The following functions prints out help (option -?) and/or warning
// messages
void GetOptionsHelp(int argc, char **argv)
{
  cerr << endl;
  cerr << "Usage:   " << endl;
  cerr << argv[0] << " -i <input file> [optional arguments]";
  cerr << endl;
  cerr << "Where a flag can be either small or capital letter and
the meaning is:  " << endl;
  cerr << '\t' << "-i <input file> is the input file." << endl;
  cerr << '\t' << "-s <system file> prints out the network
configuration." << endl;
  cerr << '\t' << "-t <Q prog file> is the Q programming solution
output file." << endl;
  cerr << '\t' << "-g <GA files base> genetic algorithm files." <<
endl;
  cerr << '\t' << "-d GA uses interval alleles." << endl;
  cerr << '\t' << "-z GA uses interval alleles with a starting zero."
<< endl;
  cerr << '\t' << "-c GA uses continuous alleles." << endl;
  cerr << '\t' << "-?  or -h Prints this message." << endl;
};

// GetOptions() parses the command line and initializes the
// data structures needed
void GetOptions(int argc, char **argv)
{
```

```
GetOpt getopt(argc, argv, "I:i:T:t:S:s:G:g:C::c::D::d::Z::z::h::?");
int option_char;
while((option_char = getopt()) != EOF)
  switch(option_char){
    // network data in Allen Wood format
  case 'I':
  case 'i':
    InputFile_str = getopt.optarg;
    break ;
    // Quadratic programming output file
  case 'T':
  case 't':
    OutTol_str = getopt.optarg;
    OutTol_str += ".tout";
    break;
    // System output in Allen Wood format
  case 'S':
  case 's':
    SysOut_str = getopt.optarg;
    break;
    // GA files
  case 'G':
  case 'g':
    GAParms_str = getopt.optarg;
    GAParms_str += "_ga.cfg";
    OutGA_str = getopt.optarg;
    OutGA_str += ".gout";
    break;
  case 'D':
  case 'd':
    interval = 1;
    break;
  case 'Z':
  case 'z':
    iZero = 1;
    break;
  case 'C':
  case 'c':
    continuous = 1;
    break;
  case '?':
  case 'h':
      GetOptionsHelp(argc, argv);
      exit(EXIT_FAILURE);
    }

};
// -*- C++ -*-
/* PFSolve.cpp - Jovan Ilic ilic@thetausa.com Wed Oct 18 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "Net.hpp"

#include <imsl.h>

bool Net::PFSolve() {
  double t2, t1 = imsl_ctime();
  int iter = 0;
  // reserve the maximum size for N-R solution
  vector<double> X(2*noBus, 0.0);
  while((pMismatch > pMaxMismatch ||
   qMismatch > qMaxMismatch) &&
  (iter < maxIter)) {
    if((iter == 0) || (qMismatch < 10.0*pMaxMismatch &&
CheckVARLimits())) {
      MakeJ();
      J.Solve(delPQ, X);
      varLimit = true;
    } else {
      MakeJ();
      J.Solve(delPQ, X);
    }
    int i, indx;
    indx = 0;
    // adjust the phases
    for(i = 0; i < buses.size(); i++) {
      if(buses[i].Type() == SWING)
  continue;
      buses[i].V(complex<double>(buses[i].V().real(), buses[i].V().imag()
```

```
+ X[indx]));
        indx++;
      }
      // adjust the magnitudes
      for(i = 0; i < buses.size(); i++) {
        if(buses[i].Type() == SWING || buses[i].Type() == GEN)
    continue;
        buses[i].V(complex<double>(buses[i].V().real()+(buses[i].V().real()*
X[indx]), buses[i].V().imag()));
        indx++;
      }
      iter++;
    }
    // update pNetInj and qNetInj arrays
    NetInj();
    return true;
}

// Checks generator var limits and if exceeded, sets the
// bus load to the appropriate limit (low or high) and
// changes the bus to LOAD type.
bool Net::CheckVARLimits() {
    int i, j, k;
    bool vlmt = false;
    for(i = 0; i < buses.size(); i++) {
      if(buses[i].Type() == GEN && buses[i].QNetInj() >
buses[i].QGenMax()){
        buses[i].QVarLimit(buses[i].QGenMax());
        buses[i].Type(LOAD);
        buses[i].BusType("LOAD");
        vlmt = true;
      }
      if(buses[i].Type() == GEN && buses[i].QNetInj() <
buses[i].QGenMin()){
        buses[i].QVarLimit(buses[i].QGenMin());
        buses[i].Type(LOAD);
        buses[i].BusType("LOAD");
        vlmt = true;
      }
    }
    return vlmt;
}
// -*- C++ -*-
```

```
/* PowerCost.cpp - Jovan Ilic ilic@thetausa.com Tue Oct 31 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

// #include <nan.h>
#include <ga/ga.h>
#include <ga/GARealGenome.h>
// #include <ga/GARealGenome.C>


#include "Net.hpp"

extern Net tst;

float PowerCostGA(GAGenome &g) {
    int i, j, k;
    float OFValue = 0.0;
    GARealGenome& genome = (GARealGenome&)g;

    // Transfer allele values to the power system object. The alleles are the
control
    // variables and can include generation and voltage output levels. Swing
bus is
    // not controlled but it must be used in the objective function calculations.

    vector<Generator> &gs = const_cast<vector<Generator>
&>(tst.GensV());
    int sbus = tst.SwingBus() - 1;

    // Power Flow from bus 4 to bus 6 seems to be reaching
    // its limit.
    vector<Bus> &bs = const_cast<vector<Bus> &>(tst.BusesV());
    vector<complex<double> > &lf_4 = bs[3].LineFlow();

    int swngIndx = 0;
    for(i = 0; i < gs.size(); i++){
      if(gs[i].GenBusNo() == tst.SwingBus()){
        swngIndx = 1;
```

```
      continue;
    }
    gs[i].GenP(genome.gene(i - swngIndx));
  }

  tst.ResetBuses();
  tst.PFSolve();
  tst.LineFlows();

  double critFlow = lf_4[2].real(); // * cos(lf_4[2].imag());

  double maxPenalty = 100.0;
  double leftNomG = 0.60; // start penalizing for less than this
  double rightNomG = 0.70; // and for greater than this value
  double dG = 0.05; // reach the max penalty at 0.6 and 0.8

  double aLeft = maxPenalty/(dG * dG);
  double bLeft = -(2.0*maxPenalty * leftNomG)/(dG * dG);
  double cLeft = (maxPenalty * leftNomG * leftNomG)/(dG * dG);

  double aRight = maxPenalty/(dG * dG);
  double bRight = -(2.0*maxPenalty*rightNomG)/(dG * dG);
  double cRight = (maxPenalty * rightNomG * rightNomG)/(dG * dG);

  double p;

  if(critFlow <= leftNomG){
    p = (aLeft * (critFlow * critFlow) + bLeft * critFlow + cLeft);
  }

  if(critFlow >= rightNomG){
    p = (aRight * (critFlow * critFlow) + bRight * critFlow + cRight);
  }

  // Choose one of these costs. This should be controlled with a
  // command line switch
  // return tst.ProdCost() + tst.VoltagePenalty();
  // return (tst.VoltagePenalty() + tst.TotalI2R());
  // return (tst.ProdCost() + tst.VoltagePenalty() + tst.TotalI2R()*500.0);
  return (tst.ProdCost() + tst.VoltagePenalty() + p);
}

extern "C" void PowerCostTolmin(int n, double *x, double *f) {
  int i, j, k;
  float OFValue = 0.0;

  // Transfer values to the power system object. The are the control
  // variables and can include generation and voltage output levels. Swing
bus is
  // not controlled but it must be used in the objective function calculations.

  vector<Generator> &gs = const_cast<vector<Generator>
&>(tst.GensV());
  int sbus = tst.SwingBus() - 1;

  int swngIndx = 0;
  for(i = 0; i < gs.size(); i++){
    if(gs[i].GenBusNo() == tst.SwingBus()){
      swngIndx = 1;
      continue;
    }
    gs[i].GenP(x[i - swngIndx]);
  }

  tst.ResetBuses();
  tst.PFSolve();
  tst.LineFlows();

  // Choose one of these consts if SQP refinement needed
  // *f = tst.ProdCost() + tst.VoltagePenalty();
  // *f = tst.VoltagePenalty() + tst.TotalI2R();
  *f = tst.ProdCost() + tst.VoltagePenalty() + tst.TotalI2R()*500.0;
}
// -*- C++ -*-
/* ProdCost.cpp - Jovan Ilic ilic@thetausa.com Sun Dec 3 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "Net.hpp"

double Net::ProdCost() {
```

```
  int i, j, k;
  double cost = 0.0;
  for(i = 0; i < gens.size(); i++)
    cost += gens[i].Cost(pBase*gens[i].GenP());

  return cost;
}

double Net::ProdCostScaled() {
  int i, j, k;
  double cost = 0.0, gCost, minCost, maxCost;
  for(i = 0; i < gens.size(); i++) {
    minCost = gens[i].Cost(gens[i].GenPMin());
    maxCost = gens[i].Cost(gens[i].GenPMax());
    gCost = gens[i].Cost(pBase*gens[i].GenP());
    cost += (gCost - minCost)/maxCost;
  }
  return cost;
}
```


D.2 GAOPF Based Unit Commitment

```
# Makefile for GAOPF Unit Commitment
# Makefile -- Jovan Ilic ilic@thetausa.com  Fri Apr 21 2000
#
# $Log$
#

.PHONY : clean


O=.o
E=
CXX=g++
CX=gcc
CXXFLAGS=-O5 -mpentiumpro
CXFLAGS=-O5 -mpentiumpro
INCD=-I/usr/local/vni/CTT3.0/include -I$(HOME)/include \
-I$(HOME)/include/stlport -I/usr/local/include
LIBD=-L$(HOME)/lib -L/usr/local/lib
LIBS=$(LINK_CNL_SHARED) -lstlport_gcc  -lm -lga

# for Windows_NT move all LIB and INCLUDE variables
# including all the ones from VCVARS32.BAT to registry
ifeq ($(OS), Windows_NT)
O=.obj
E=.exe
CXX=cl
CX=cl
CXXFLAGS=/nologo /TP /GR /MT /G6  /GX /O2 /D "WIN32"
CXFLAGS=/nologo /G6 /MT /O2 /D "WIN32"
INCD=
LIBD=/link /nodefaultlib:libc.lib /nologo /subsystem:console \
/incremental:no
LIBS=dspgen.obj cmath_s.lib ga.lib kernel32.lib user32.lib
endif

.SUFFIXES: .cpp .c .CPP .C .CC .CXX $(O)


%$(E) : %.cpp
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.cc
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.C
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.CC
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.CXX
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(O) : %.CXX
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(O) : %.cpp
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(O) : %.cc
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@
```

```
%$(O) : %.C
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(O) : %.CC
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(E) : %.c
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(O) : %.c
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

# insert source files here

CXXSRC=main.cpp Net.cpp GetOpt.cpp options.cpp Bus.cpp \
Generator.cpp Branch.cpp MatrixSPD.cpp MakeGB.cpp \
MakeJac.cpp  UCobjective.cpp PFSolve.cpp LineFlows.cpp \
PowerCost.cpp ProdCost.cpp

CSRC=

OBJ := $(patsubst %.c, %$(O), $(CSRC)) \
$(patsubst %.cpp, %$(O), $(CXXSRC))


# define the target(s)

all : ./data/gauc$(E)

./data/gauc$(E) : $(OBJ)
$(CXX) -o $@ /home/ilic/lib/dspgen.o $(OBJ) $(LIBD) $(LIBS)

clean:
rm $(OBJ) ./data/gauc$(E)

echo :
@echo $(OBJ)

depend :
@makedepend -Y -I. $(CSRC) $(CXXSRC) -f depend.inc

# this file needs to exist before running make.
# "touch depend.inc" will do it
include depend.inc

# DO NOT DELETE

main.o: main.cpp stds.hpp GetOpt.h externs.hpp Net.hpp \
 Bus.hpp  Generator.hpp Branch.hpp MatrixSPD.hpp
Net.o: Net.cpp Net.hpp stds.hpp Bus.hpp Generator.hpp \
 Branch.hpp MatrixSPD.hpp externs.hpp
GetOpt.o: GetOpt.cpp GetOpt.h
options.o: options.cpp stds.hpp GetOpt.h externs.hpp
Bus.o: Bus.cpp Bus.hpp stds.hpp Generator.hpp Branch.hpp
Generator.o: Generator.cpp Generator.hpp stds.hpp
Branch.o: Branch.cpp Branch.hpp stds.hpp
MatrixSPD.o: MatrixSPD.cpp MatrixSPD.hpp
MakeGB.o: MakeGB.cpp Net.hpp stds.hpp Bus.hpp Generator.hpp \
 Branch.hpp  MatrixSPD.hpp externs.hpp
MakeJac.o: MakeJac.cpp Net.hpp stds.hpp Bus.hpp Generator.hpp \
 Branch.hpp MatrixSPD.hpp externs.hpp
UCobjective.o: UCobjective.cpp Net.hpp stds.hpp Bus.hpp \
 Generator.hpp Branch.hpp MatrixSPD.hpp externs.hpp
PFSolve.o: PFSolve.cpp Net.hpp stds.hpp Bus.hpp Generator.hpp \
 Branch.hpp MatrixSPD.hpp externs.hpp
LineFlows.o: LineFlows.cpp Net.hpp stds.hpp Bus.hpp \
 Generator.hpp Branch.hpp MatrixSPD.hpp externs.hpp
PowerCost.o: PowerCost.cpp Net.hpp stds.hpp Bus.hpp \
 Generator.hpp Branch.hpp MatrixSPD.hpp externs.hpp
ProdCost.o: ProdCost.cpp Net.hpp stds.hpp Bus.hpp Generator.hpp \
 Branch.hpp MatrixSPD.hpp externs.hpp
```

```
// -*- C++ -*-
/* externs.hpp - Jovan Ilic ilic@thetausa.com Mon Oct 4 1999
 *
 * $Header$
 * $Log$
 */

#ifndef _EXTERNS_HPP
#define _EXTERNS_HPP

extern string InputFile_str;
extern string GAParms_str, ErrsGA_str, ucIn_str, ucOut_str;
extern vector<double> ucLoad;
```

```
extern int latex;

extern string gaStart_str;

extern void GetOptionsHelp(int, char **);
extern void GetOptions(int, char **);



// Microsoft VC++ specific
#ifdef _MSC_VER

#define M_PI 3.14159265358979323846

#endif // _MSC_VER


#endif
// -*- C++ -*-
/* Generator.hpp - Jovan Ilic ilic@thetausa.com Tue Sep 5 2000
 *
 * $Header$
 * $Log$
 */

#ifndef _GENERATOR_HPP
#define _GENERATOR_HPP

#include "stds.hpp"

#include <math.h>

class Generator {
private: // make clear that private part is first by default

  enum {GEN = 2, LOAD, GEN_HIGHVAR, GEN_LOWVAR, GEN_REG,
SWING};

  int genBusNo, tp;
  double genP, genV, genQMax, genQMin;

  // cost function calculation additions
  // This could also be a line segment description
  vector<double> costPoly; // poly coefficients in increasing order
  double genPMin, genPMax; // generation limits

  // UC variables
  int runTime, minDownTime, minUpTime;
  bool on;

protected: // protected members


public: // public members

  Generator(){ // default constructor
    genBusNo = -1;
    tp = GEN;
    genP = genV = genQMax = genQMin = 0.0;
    genPMin = genPMax = 0.0;

    runTime = -2; minDownTime = 2;
    minUpTime = 2;

    on = false;

  }
  Generator(const Generator &g){ // copy constructor
    genBusNo = g.GenBusNo();
    tp = g.Type();
    genP = g.GenP();
    genV = g.GenV();
    genQMax = g.GenQMax();
    genQMin = g.GenQMin();
    genPMin = g.GenPMin();
    genPMax = g.GenPMax();
    costPoly = const_cast<Generator &>(g).CostPolyV();

    runTime = g.RunTime();
    minUpTime = g.MinUpTime();
    minDownTime = g.MinDownTime();
    on = g.On();

  }
  Generator &operator=(const Generator &g){ // overloaded = operator
    genBusNo = g.GenBusNo();
```

```
    tp = g.Type();
    genP = g.GenP();
    genV = g.GenV();
    genQMax = g.GenQMax();
    genQMin = g.GenQMin();
    genPMin = g.GenPMin();
    genPMax = g.GenPMax();
    costPoly = const_cast<Generator &>(g).CostPolyV();

    runTime = g.RunTime();
    minUpTime = g.MinUpTime();
    minDownTime = g.MinDownTime();

    on = g.On();

    return *this;

  }

  virtual ~Generator(){}; // virtual destructor

  int GenBusNo() const {return genBusNo;}
  void GenBusNo(const int g) {genBusNo = g;}

  double GenP() const { return genP;}
  void GenP(const double gp) {genP = gp;}

  double GenV() const { return genV;}
  void GenV(const double gv) { genV = gv;}

  double GenQMax() const {return genQMax; }
  void GenQMax(const double gq) {genQMax = gq;}

  double GenQMin() const { return genQMin;}
  void GenQMin(const double gq) { genQMin = gq; }

  double GenPMax() const { return genPMax;}
  void GenPMax(const double gp) { genPMax = gp;}

  double GenPMin() const {return genPMin;}
  void GenPMin(const double gp) {genPMin = gp;}

  // save (in order) a polynomial cost coefficient
  void CostPoly(const double coef) {
    costPoly.push_back(coef);
  }

  // whole cost vector assignment and retrival
  const vector<double> &CostPolyV() { return costPoly; }
  void CostPoly(const vector<double> &c) { costPoly = c;}

  // for Cost() and CostInc(), argumment should be between
  // genPMin and genPMax. p should be non-pu power
  // genPMin and genPMax are also not scaled
  double Cost(double p) const {

    double c = 0.0;


    // generation limits are here only because of the swing bus.
    if(p != 0.0) {
      if(p < genPMin)
  return 4.0*Cost(genPMax);

      if(p > genPMax)
  return 4*Cost(genPMax);
    }

    for(int i = 0; i < costPoly.size(); i++)
      c += costPoly[i] * pow(p, static_cast<double>(i));


    return c;
  }

  double CostInc(double p) {

    double c = 0.0;

    if(p < genPMin)
      return CostInc(genPMin);

    if(p > genPMax)
      return CostInc(genPMax);

    for(int i = 1; i < costPoly.size(); i++)
```

```
      c += costPoly[i] * i * pow(p, static_cast<double>(i - 1));

    return c;
  }


  int Type() const { return tp;}
  void Type(const int t) {
    if((t <= SWING) && (t >= GEN))
      tp = t;
  }

  int RunTime() const { return runTime;}
  void RunTime(const int rt) { runTime = rt;}

  int MinUpTime() const { return minUpTime;}
  void MinUpTime(const int ut) { minUpTime = ut;}

  int MinDownTime() const { return minDownTime; }
  void MinDownTime(const int dt) { minDownTime = dt; }

  bool On() const { return on;}
  void On(const bool o) {on = o;}

  friend ostream &operator<<(ostream &, const Generator &);

};

#endif
// -*- C++ -*-
/* main.cpp - Jovan Ilic ilic@thetausa.com Fri Apr 21 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include <ga/ga.h>
#include <ga/GARealGenome.h>
#include <ga/GARealGenome.C>

#include "stds.hpp"
#include "GetOpt.h"

#include <limits>

#define extern
#include "externs.hpp"
#undef extern

#include <imsl.h>

#include "Net.hpp"

Net tst;

extern float PowerCostGA(GAGenome &);
extern float Objective(GAGenome &g);

int main(int argc, char **argv) {

  enum {GEN = 2, LOAD, GEN_HIGHVAR, GEN_LOWVAR, GEN_REG,
SWING};

  int i, j, k; // , activeCount;

  const int sampleNo = 10;

  float totCost;
  float pBase;

  InputFile_str = "none";
  GAParms_str = "none";
  ErrsGA_str = "none";
  ucIn_str = "none";
  ucOut_str = "none";
  gaStart_str = "none";

  latex = 0;

  fstream cfgGA_fs, errsGA_fs;
  fstream inputFile_fs, ucIn_fs, ucOut_fs;

  GetOptions(argc, argv);
```

```
  if(InputFile_str == "none"){
    cerr << "System description file must be available for unit
commitment" << endl;
    GetOptionsHelp(argc, argv);
    exit(-1);
  }else{
    inputFile_fs.open(InputFile_str.c_str(), ios::in);
    if(!inputFile_fs){
      cerr << "Could not open " << InputFile_str << " for reading" <<
endl;
      exit(-1);
    }
    tst.ReadAW(InputFile_str);
    tst.MakeY();
  }


  if(ucOut_str != "none"){
    ucOut_fs.open(ucOut_str.c_str(), ios::out);
    if(!ucOut_fs.is_open()){
      cerr << "Could not open " << ucOut_str << " for writing" << endl;
      exit(-1);
    }
  }

  if(GAParms_str != "none") {
    cfgGA_fs.open(GAParms_str.c_str(), ios::in);
    if(!cfgGA_fs) {
      cerr << "Cannot open:  " << GAParms_str << " for reading" <<
endl;
      exit(-1);
    }
  }


  // read in the load schedule
  double ld;
  while(ucIn_fs >> ld){
    ucLoad.push_back(ld);
  };
```

```
  vector<Generator> &gns = const_cast<vector<Generator >
&>(tst.GensV());

  vector<Bus> &bs = const_cast<vector<Bus> &>(tst.BusesV());

  // set up the GALIB routines. SWING gen is always on
  GA1DBinaryStringGenome genome(tst.GensV().size() - 1, Objective);
  gns[tst.SwingBus() - 1].On(true);

  GASteadyStateGA ga(genome);

  GAParameterList params;
  GASimpleGA::registerDefaultParameters(params);

  if(GAParms_str != "none") {
    params.read(GAParms_str.c_str());
  } else {
    params.set(gaNnGenerations, 20);
    params.set(gaNpopulationSize, 20);
    params.set(gaNscoreFrequency, 1);
    params.set(gaNflushFrequency, 10);
    params.set(gaNselectScores, 4);
    params.set(gaNminimaxi, -1);
    ga.set(gaNscoreFilename, "uc.plt");
  }

  ga.parameters(params);
  ga.minimize();

  ga.set(gaNscoreFilename, "uc.plt");

  vector<double> nominalLoad;
  vector<double> rx;

  // save nominal P loads read from the config file
  for(i = 0; i < bs.size(); i++){
    nominalLoad.push_back(bs[i].PLoad());
    rx.push_back(0.0);
  }

  for(i = 0; i < sampleNo; i++){

    ga.evolve();
```

```
    GA1DBinaryStringGenome &gen = (GA1DBinaryStringGenome
&)ga.statistics().bestIndividual();

    GARealAlleleSetArray GAUCAlleles;

    for(j = 0; j < GAUCAlleles.size(); j++)
      GAUCAlleles.remove(j);

    // setup variable constraints
    int swngIndx = 0;
    pBase = tst.PBase();
    for(j = 0; j < gns.size(); j++) {
      if(gns[j].GenBusNo() == tst.SwingBus()) {
  swngIndx = 1;
  continue;
      }
      if(gen[j - swngIndx])
  GAUCAlleles.add(gns[j].GenPMin()/pBase, gns[j].GenPMax()/pBase,
      GAAllele::INCLUSIVE,GAAllele::INCLUSIVE );

    }

    GARealGenome GAUCGenome(GAUCAlleles, PowerCostGA);

    GAParameterList parms;
    GASteadyStateGA::registerDefaultParameters(parms);
    GASteadyStateGA gauc(GAUCGenome);

    if(gaStart_str != "none"){
      parms.read(gaStart_str.c_str());
    } else {

      parms.set(gaNnGenerations, 30);
      parms.set(gaNpopulationSize, 20);
      parms.set(gaNscoreFrequency, 1);
      parms.set(gaNflushFrequency, 10);
      parms.set(gaNselectScores, 4);
      parms.set(gaNminimaxi, -1);
      gauc.set(gaNscoreFilename, "refine.plt");
    }

    gauc.parameters(parms);
    gauc.scaling(GANoScaling());

    gauc.evolve();

    // Save the "optimal" generation solution
    GARealGenome& g =
(GARealGenome&)gauc.statistics().bestIndividual();

    int indx = 0;
    swngIndx = 0;
    for(j = 0; j < gns.size(); j++){

      gns[j].GenP(0.0);

      if(gns[j].GenBusNo() == tst.SwingBus()){
  swngIndx = 1;
  continue;
      }

      if(gen[j - swngIndx]){
  gns[j].On(true);
  bs[gns[j].GenBusNo() - 1].Type(GEN);
  gns[j].GenP(g[indx++]);
      } else {
  gns[j].On(false);
  bs[gns[j].GenBusNo() - 1].Type(LOAD);
      }
    }

    tst.ResetBuses();
    tst.PFSolve();
    tst.LineFlows();



    int intSched;
    intSched = 0;

    for(j = 0; j < gen.length(); j++)
      intSched += pow(2,j) * static_cast<int>(gen[j]);

    float totGen;
```

```
    if(ucOut_fs.is_open()){
      if(latex){
for(j = 0; j < gen.length(); j++) {
  if(gen[j])
    ucOut_fs << "On & ";
  else
    ucOut_fs << "Off & ";
}
ucOut_fs << tst.PLoadTotal() << " & ";
ucOut_fs << tst.PGenTotal() << " & ";
ucOut_fs << tst.ProdCost() << " & ";
ucOut_fs << " \\\\ \\hline" << endl;
    } else {
ucOut_fs << (i+1) << ",\t";
// print out schedule
ucOut_fs << intSched << ",\t";
// print out bus loads
for(j = 0; j < bs.size(); j++)
  ucOut_fs << bs[j].PLoad() << ",\t";
ucOut_fs << endl;
    }
  } else {
    if(latex){
for(j = 0; j < gen.length(); j++) {
  if(gen[j])
    cout << "On & ";
  else
    cout << "Off & ";
}
cout << tst.PLoadTotal() << " & ";
cout << tst.PGenTotal() << " & ";
cout << tst.ProdCost() << " & ";
cout << " \\\\ \\hline" << endl;
    }else {


cout << (i+1) << ",\t";
// print out schedule
cout << intSched << ",\t";
// print out bus loads
for(j = 0; j < bs.size(); j++)
  cout << bs[j].PLoad() << ",\t";
cout << endl;
    }
  }

  // construct next load sample

  imsl_d_random_uniform(bs.size(), IMSL_RETURN_USER, &rx[0],0);

  for(j = 0; j < bs.size(); j++)
    bs[j].PLoad(rx[j] + 0.1);

}


if(cfgGA_fs.is_open()){
  cfgGA_fs.close();
}

if(errsGA_fs.is_open())
  errsGA_fs.close();

if(inputFile_fs.is_open())
  inputFile_fs.close();

if(ucIn_fs.is_open())
  ucIn_fs.close();

if(ucOut_fs.is_open())
  ucOut_fs.close();

  return 0;
}
// -*- C++ -*-
/* Net.cpp - Jovan Ilic ilic@thetausa.com Tue Sep 5 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "Net.hpp"
```

```
extern bool lessTB(const Branch &, const Branch &);
extern bool lessFB(const Branch &, const Branch &);
extern bool lessGen(const Generator &, const Generator &);
extern bool lessBus(const Bus &, const Bus &);

bool Net::ReadAW(const string &f) {

  char tmpLn[512];
  int i, tmpI;
  double tmpD;
  string tmpStr;

  fstream fs(f.c_str(), ios::in);

  if(!fs) {
    cerr << "Could not open:  " << f << " for reading." << endl;
    exit(-1);
  }

  // Get two title lines
  fs.getline(tmpLn, 512); Title1 = tmpLn;
  fs.getline(tmpLn, 512); Title2 = tmpLn;

  // read number of buses, number of generators
  // number of branches, and swing bus number
  fs >> noBus >> noGen >> noBr >> swingBus >> noCap;

  // read bus data
  for(i = 0; i < noBus; i++) {
    buses.push_back(Bus());
    fs >> tmpI; buses[i].BusNo(tmpI);
    fs >> tmpStr; buses[i].BusName(tmpStr);
    fs >> tmpD; buses[i].PLoad(tmpD);
    fs >> tmpD; buses[i].QLoad(tmpD);
    fs >> tmpD; buses[i].VLL(tmpD);
    fs >> tmpD; buses[i].VHL(tmpD);
    buses[i].BusType("LOAD"); // default
    buses[i].Type(LOAD); // default
  }

  sort(buses.begin(), buses.end(), lessBus);

  // read in generator data
  for(i = 0; i < noGen; i++) {
    gens.push_back(Generator());
    fs >> tmpI; gens[i].GenBusNo(tmpI);
    buses[tmpI-1].BusType("GEN"); buses[tmpI-1].Type(GEN);
    fs >> tmpD; gens[i].GenP(tmpD);
    fs >> tmpD; gens[i].GenV(tmpD);
    fs >> tmpD; gens[i].GenQMax(tmpD);
    fs >> tmpD; gens[i].GenQMin(tmpD);
    fs >> tmpD; gens[i].GenPMin(tmpD);
    fs >> tmpD; gens[i].GenPMax(tmpD);
    // get the rest of the line which contains cost
    // coefficients in increasing order
    fs.getline(tmpLn, 512);
    istrstream tmpStr(tmpLn);
    while(tmpStr){
      tmpStr >> tmpD;
      if(tmpStr)
gens[i].CostPoly(tmpD);
    }
  }

  buses[swingBus-1].BusType("SWING");
  buses[swingBus-1].Type(SWING);

  sort(gens.begin(), gens.end(), lessGen);

  // this must be done after sorting gens;
  for(i = 0; i < gens.size(); i++){
    int busIndx;
    busIndx = gens[i].GenBusNo() - 1;
    buses[busIndx].Gen(&gens[i]); // attach generator to bus
    buses[busIndx].V(gens[i].GenV());
  }

  // read in branches data
  noTran = 0;
  trans.erase(trans.begin(), trans.end());
  for(i = 0; i < noBr; i++) {
    branches.push_back(Branch());
    fs >> tmpI; branches[i].BrFB(tmpI);
    fs >> tmpI; branches[i].BrTB(tmpI);
    fs >> tmpD; branches[i].BrR(tmpD);
```

```
    fs ≫ tmpD; branches[i].BrX(tmpD);
    fs ≫ tmpD; branches[i].BrBcap(tmpD);
    fs ≫ tmpD; branches[i].BrLineFlowLimit(tmpD);
    fs ≫ tmpLn; branches[i].BrLineType(tmpLn);
    if(branches[i].BrLineType() == "TRAN"){
      noTran++;
      trans.push_back(tmpI);
      fs ≫ tmpI; branches[i].BrTapBus(tmpI);
      fs ≫ tmpI; branches[i].BrNonTapBus(tmpI);
      fs ≫ tmpD; branches[i].BrTapRatio(tmpD);
    }
  }

  sort(trans.begin(), trans.end());

  // order branches according to "from node" first ....
  sort(branches.begin(), branches.end(), lessFB);

  vector<Branch>::iterator cur, left;

  left = branches.begin();

  // then according to "to node".
  for(cur = branches.begin(); cur < branches.end(); cur++){
    if((*left).BrFB() != (*cur).BrFB()){
      sort(left, cur, lessTB);
      left = cur;
    }
  }


  // Get the capacitors in the bus data;
  caps.erase(caps.begin(), caps.end());
  caps.reserve(noCap);
  for(i = 0; i < noCap; i++){
    fs ≫ tmpI ≫ tmpD;
    caps.push_back(tmpI);
    buses[tmpI - 1].YSh(tmpD);
  }

  sort(caps.begin(), caps.end());

  // this is good place to finalize Bus::adjBranch
    // Bus::adjBranch shold be erased here first
    // The destination bus read from a Branch is either
    // Branch::BrFB() or Branch::BrTB() depending on which
    // which one is the current bus (== "bus number" + 1)
    // This is probably an issue with tap transformers only.
    for(i = 0; i < branches.size(); i++) {
      buses[branches[i].BrFB()-1].AdjBranch(&branches[i]);
      buses[branches[i].BrTB()-1].AdjBranch(&branches[i]);

    }

    // Tracing the adjacent buses should make programming really
    // easier and future exapansion simpler.
    for(i = 0; i < buses.size(); i++){
      vector<Branch *> tmpBr = buses[i].AdjBranch();
      for(int j = 0; j < tmpBr.size(); j++){
        if(tmpBr[j]→BrFB() == i+1)
    buses[i].AdjBus(tmpBr[j]→BrTB());
        else
    buses[i].AdjBus(tmpBr[j]→BrFB());
        buses[i].LineFlow().push_back(0);
      }
    }

    return true;
  }

  bool Net::ResetBuses() {

    for(int i = 0; i < buses.size(); i++) {
      if(buses[i].Type() != SWING)
        if(buses[i].Gen() != NULL){
    buses[i].V(buses[i].Gen()→GenV());
    buses[i].QVarLimit(0.0);

    if(buses[i].Gen()→On())
      buses[i].Type(GEN);
    else // Generator not on
      buses[i].Type(LOAD);

    buses[i].BusType("GEN");

        } else {
```

```
    buses[i].V(1.0);
    buses[i].QVarLimit(0.0);
    buses[i].Type(LOAD);
    buses[i].BusType("LOAD");
        }
    }
    PMismatch(2.0*PMaxMismatch());
    QMismatch(2.0*QMaxMismatch());

    return true;
}

bool Net::PrintAW(ostream &os) {

    int i;

    ios::fmtflags oldFlags = os.flags();

    os << endl << endl << "Input File Name:  " << InputFile_str << endl <<
endl << endl;
    os << Title1 << endl;
    os << Title2 << endl;

    os << " Number of Buses = " << "\t\t" << noBus << endl;
    os << " Number of Lines = " << "\t\t" << noBr << endl;
    os << " Number of Generators = " << "\t" << noGen << endl;
    os << " Number of Transformers = " << "\t" << noTran << endl;
    os << " Swing Bus at bus number " << "\t" << swingBus << endl;
    os << endl << "Bus Data" << endl << endl;
    os << "Bus Voltage Voltage Bus " << endl;
    os << "Number Bus Name Mw Load Mvar Load High Limit Low limit
Type " << endl;
    os << "------ -------- ---------- ---------- ----------
---------- -----" << endl;

    for(i = 0; i < noBus; i++)
      os << buses[i];

    os << endl;

    os << "Generator Data" << endl << endl;
    os << "Bus Mw Sched H Mvar L MVar Low P High P C B A " << endl;
    os << "number Gen.  Volt.  Limit Limit Limit Limit " << endl;
    os << "------ ----- ------ ------- ------- ------ ------- -------
------- -------" << endl;

    for(i = 0; i < noGen; i++)
      os << gens[i];

    os << endl;
    os << "Branch Data" << endl;
    os << endl;

    os << "From To Branch Branch Line Branch Branch Tap Non-Tap Tap "
<< endl;
    os << "Bus Bus R X Charging Mva Limit Type Bus Bus Ratio " << endl;
    os << "---- ---- ------ ------ -------- --------- ------ ----
------- ------" << endl;

    for(i = 0; i < noBr; i++)
      os << branches[i];

    os.setf(ios::left, ios::adjustfield);

    // List of capacitors with bus indecies here.
    os << endl << "Capacitors List" << endl << endl;
    os << "Bus Number " << "Capacitor Value" << endl;
    os << "---------- " << "---------------" << endl;
    for(i = 0; i < noCap; i++) {
      os << setw(10) << setprecision(5) << caps[i] << " " <<
buses[caps[i]-1].YSh() << endl;
    }

    os.flags(oldFlags);
    return true;
}


bool Net::PrintAWOut(ostream &os) {

    int i,j,k,fb,tb;

    double r2d = 180.0/M_PI;

    ios::fmtflags oldFlags = os.flags();
```

```
  os.setf(ios::right, ios::adjustfield);

  os.setf(ios::fixed, ios::floatfield);

  os <<
"=========================================";
  os <<
"=========================================" <<
endl;
  os << " Solution Output " << endl;
  os <<
"=========================================";
  os <<
"=========================================" <<
endl;

  os << endl;

  os << "Load Flow Solution:" << endl;
  os << endl;
  os << "file:  " << InputFile_str << endl;
  os << endl;
  os << Title1 << endl;
  os << Title2 << endl;
  os << endl;
  os << "From Volt.  Volt.  Mw Mvar Mw Mvar To Mw Mvar Mva " << endl;
  os << "Bus Mag.  Angle Load Load Gen.  Gen.  Bus Flow Flow Flow "
<< endl;
  os << "---- ----- ------- ------- ------- ------- ------- ----
------- ------- -------" << endl;

  os << endl;

  for(i = 0; i < buses.size(); i++) {
    os << endl;
    os << setw(4) << (i + 1) << " ";
    os << setw(5) << setprecision(3) << buses[i].V().real() << " ";
    os << setw(7) << setprecision(3) << (r2d*buses[i].V().imag()) << " ";
    os << setw(7) << setprecision(2) << (pBase*buses[i].PLoad()) << " ";
    os << setw(7) << setprecision(2) << (pBase*buses[i].QLoad()) << " ";

    if(buses[i].Gen() != NULL){
      os << setw(7) << setprecision(2) << (pBase*buses[i].PGen()) << " ";
      os << setw(7) << setprecision(2) << (pBase*buses[i].QGen()) << " ";
    } else {
      os << setw(7) << setprecision(2) << 0.0 << " ";
      os << setw(7) << setprecision(2) << 0.0 << " ";
    }

    os << endl;
    const vector<int> &tmpBus = buses[i].AdjBus();
    const vector<complex<double> > &lf = buses[i].LineFlow();
    for(j = 0; j < tmpBus.size(); j++) {
      for (k = 1; k <= 51; k++)
  os << ' ';
      os << setw(4) << tmpBus[j] << " ";
      os << setw(7) << setprecision(2) << (pBase*lf[j].real()) << " ";
      os << setw(7) << setprecision(2) << (pBase*lf[j].imag()) << " ";
      os << setw(7) << setprecision(2) << (pBase*abs(lf[j])) << endl;
    }
  }

  os << endl << endl;
  os << "Mw gen = " << setw(7) << setprecision(3) << (pBase*pGenTotal)
<< endl;
  os << "Mvar gen = " << setw(7) << setprecision(3) << (pBase*qGenTotal)
<< endl;
  os << "Mw load = " << setw(7) << setprecision(3) << (pBase*pLoadTotal)
<< endl;
  os << "Mvar load = " << setw(7) << setprecision(3) <<
(pBase*qLoadTotal) << endl;
  os << "Total I2R Mw losses = " << setw(7) << setprecision(3) <<
(pBase*totalI2R) << endl;
  os << "Total I2X Mvar losses = " << setw(7) << setprecision(3) <<
(pBase*totalI2X) << endl;
  os << "Total charging Mvar = " << setw(7) << setprecision(3) <<
(pBase*totalCharging) << endl;


  os <<
"=========================================";
  os <<
"========================================="
<< endl;
  os << " End of Output " << endl;
  os <<
```

```
"=========================================";
  os <<
"=========================================="
<< endl;

  os.flags(oldFlags);

  return true;
}
/* - J.Ilic e:/ilic/nn/pnn/specht/options.cc
 * Wed Oct 14 1992
 * <ilic@morse.eece.maine.edu>
 */

// $Log$

#ifndef lint
static char *rcsid = "$Id";
#endif

#include "stds.hpp"

#include "GetOpt.h"

#include "externs.hpp"


// The following functions prints out help (option -?) and/or warning
// messages
void GetOptionsHelp(int argc, char **argv)
{
  cerr << endl;
  cerr << "Usage:   " << endl;
  cerr << argv[0] << " -i <input file> [optional arguments]";
  cerr << endl;
  cerr << "Where a flag can be either small or capital letter and
the meaning is:  " << endl;
  cerr << '\t' << "-i <input file> is the input file." << endl;
  cerr << '\t' << "-g <GA parms files> genetic parameters files." <<
endl;
  cerr << '\t' << "-o <UC schedule> unit commitment schedule." <<
endl;
  cerr << '\t' << "-s <GAlib parms file> use GA start for SQP." <<
endl;
  cerr << '\t' << "-l output schedule as LaTeX table." << endl;
  cerr << '\t' << "-?  or -h Prints this message." << endl;
};

// GetOptions() parses the command line and initializes the
// data structures needed
void GetOptions(int argc, char **argv)
{

  GetOpt getopt(argc, argv, "S:s:I:i:G:g:L::l::O:o:h:?");
  int option_char;
  while((option_char = getopt()) != EOF)
    switch(option_char){
      // network data in Allen Wood format
    case 'I':
    case 'i':
      InputFile_str = getopt.optarg;
      break ;
      // binary string GA parameters
    case 'G':
    case 'g':
      GAParms_str = getopt.optarg;
      break;
      // unit commitment output files
    case 'O':
    case 'o':
      ucOut_str = getopt.optarg;
      break;
      // real variable GA parameters
    case 'S':
    case 's':
      gaStart_str = getopt.optarg;
      break;
    case 'l':
    case 'L':
      latex = 1;
      break;
    case '?':
    case 'h':
      GetOptionsHelp(argc, argv);
      exit(EXIT_FAILURE);
    }
```

```
};
// -*- C++ -*-
/* PowerCost.cpp - Jovan Ilic ilic@thetausa.com Tue Oct 31 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

// #include <nan.h>
#include <ga/ga.h>
#include <ga/GARealGenome.h>
// #include <ga/GARealGenome.C>


#include "Net.hpp"

extern Net tst;

float PowerCostGA(GAGenome &g) {

  int i, j, k;

  float OFValue = 0.0;

  GARealGenome& genome = (GARealGenome&)g;

  // cout << g << endl;

  // Transfer allele values to the power system object. The alleles are the
control
  // variables and can include generation and voltage output levels. Swing
bus is
  // not controlled but it must be used in the objective function calculations.

  vector<Generator> &gs = const_cast<vector<Generator>
&>(tst.GensV());
  int sbus = tst.SwingBus() - 1;

  int swngIndx = 0;
  int indx = 0;

  for(i = 0; i < gs.size(); i++){
    gs[i].GenP(0.0);

    if(gs[i].GenBusNo() == tst.SwingBus()){
      swngIndx = 1;
      continue;
    }

    if(gs[i].On()){
      gs[i].GenP(genome.gene(indx++));
    }
  }

  tst.ResetBuses();
  tst.PFSolve();
  tst.LineFlows();

  return tst.ProdCost() + tst.VoltagePenalty();
  // return (tst.VoltagePenalty() + tst.TotalI2R());
  // return (tst.ProdCost() + tst.VoltagePenalty() + tst.TotalI2R()*100.0);

}


extern "C" void PowerCostTolmin(int n, double *x, double *f) {

  int i, j, k;

  float OFValue = 0.0;

  // Transfer values to the power system object. The are the control
  // variables and can include generation and voltage output levels. Swing
bus is
  // not controlled but it must be used in the objective function calculations.

  vector<Generator> &gs = const_cast<vector<Generator>
&>(tst.GensV());
  int sbus = tst.SwingBus() - 1;

  int swngIndx = 0;
  for(i = 0; i < gs.size(); i++){
    if(gs[i].GenBusNo() == tst.SwingBus()){
      swngIndx = 1;
```

```
      continue;
    }
    gs[i].GenP(x[i - swngIndx]);
  }

  tst.ResetBuses();
  tst.PFSolve();
  tst.LineFlows();

  // *f = tst.ProdCost() + tst.VoltagePenalty();
  // *f = tst.VoltagePenalty() + tst.TotalI2R();
  *f = tst.ProdCost() + tst.VoltagePenalty() + tst.TotalI2R();

}
// -*- C++ -*-
/* ProdCost.cpp - Jovan Ilic ilic@thetausa.com Sun Dec 3 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "Net.hpp"


double Net::ProdCost() {

  int i, j, k;

  double cost = 0.0;

  for(i = 0; i < gens.size(); i++)
    if(gens[i].On())
        cost += gens[i].Cost(pBase * gens[i].GenP());

  return cost;
}

double Net::ProdCostScaled() {

  int i, j, k;

  double cost = 0.0, gCost, minCost, maxCost;

  for(i = 0; i < gens.size(); i++) {
    minCost = gens[i].Cost(gens[i].GenPMin());
    maxCost = gens[i].Cost(gens[i].GenPMax());
    gCost = gens[i].Cost(pBase*gens[i].GenP());

    cost += (gCost - minCost)/maxCost;
  }

  return cost;

}
```

D.3 Decision Tree Machine Learning

```
# Makefile for Binary tree induction
# Makefile  -- Jovan Ilic ilic@thetausa.com  Mon Apr 24 2000
#
# $Log$
#

.PHONY : clean


O=.o
E=
CXX=g++
CX=gcc
CXXFLAGS= -mpentiumpro  -O5  -fstrict-aliasing
CXFLAGS= -mpentiumpro -O5
INCD=-I$(HOME)/include -I$(HOME)/ilic/include/stlport \
-I/usr/local/include
LIBD=-L$(HOME)/lib -L/usr/local/lib
LIBS=-lstlport_gcc -lm

ifeq ($(OS), Windows_NT)
O=.obj
E=.exe
```

```
CXX=cl
CX=cl
CXXFLAGS=/nologo /G6 /GX -DDEBUG
CXFLAGS=/nologo /G6 /GX -DDEBUG
INCD=-I$(HOME)/include -I/msdev/vc98/include
LIBD=-L$(HOME)/LIB -I/msdev/vc98/lib
LIBS=
endif

.SUFFIXES: .cpp .c .CPP .C .CC .CXX $(O)


%$(E) : %.cpp
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.cc
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.C
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.CC
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(E) : %.CXX
$(CXX) $(CXXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(O) : %.CXX
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(O) : %.cpp
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(O) : %.cc
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(O) : %.C
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(O) : %.CC
$(CXX) $(CXXFLAGS) $(INCD) -c $< -o $@

%$(E) : %.c
$(CX) $(CXFLAGS) $(INCD) $(LIBD) $(LIBS) -o $@ $<

%$(O) : %.c
$(CX) $(CXFLAGS) $(INCD) -c $< -o $@

# insert source files here

CXXSRC = GetOpt.cpp options.cpp DTree.cpp BNode.cpp \
ParseData.cpp ParseNames.cpp GetToken.cpp main.cpp \
SplitID3.cpp Train.cpp PrintCTree.cpp Pst.cpp \
PrintRules.cpp

CXXSRC1 = GetOpt.cpp options.cpp DTree.cpp BNode.cpp \
ParseData.cpp ParseNames.cpp GetToken.cpp maintest.cpp \
SplitID3.cpp Train.cpp PrintCTree.cpp Pst.cpp \
PrintRules.cpp dTree.cpp

CXXSRC2 = GetOpt.cpp options.cpp DTree.cpp BNode.cpp \
ParseData.cpp ParseNames.cpp GetToken.cpp mainlim.cpp \
SplitID3.cpp Train.cpp PrintCTree.cpp Pst.cpp \
PrintRules.cpp


CSRC=

OBJ := $(patsubst %.c, %$(O), $(CSRC)) $(patsubst %.cpp, %$(O), $(CXXSRC))

OBJ1 := $(patsubst %.c, %$(O), $(CSRC)) $(patsubst %.cpp, %$(O), $(CXXSRC1))

OBJ2 := $(patsubst %.c, %$(O), $(CSRC)) $(patsubst %.cpp, %$(O), $(CXXSRC2))


# define the target(s)

all : data/bt$(E) data/testtree$(E) data/mainlim$(E)

data/bt$(E) : $(OBJ)
$(CXX)  -o $@ $(OBJ) $(LIBD) $(LIBS)

data/testtree$(E) : $(OBJ1)

data/mainlim$(E) : $(OBJ2)
$(CXX)  -o $@ $(OBJ2) $(LIBD) $(LIBS)
```

```
clean:
rm $(OBJ) data/bt$(E) data/testtree$(E) data/mainlim$(E) \
mainlim$(O) maintest$(O) dTree$(O)

echo :
@echo $(OBJ)

depend :
@gcc -MM -I$(INCD) $(CSRC) $(CXXSRC) maintest.cpp \
mainlim.cpp > depend.inc

# this file needs to exist before running make.
# "touch depend.inc" will do it
include depend.inc

# DO NOT DELETE

GetOpt.o: GetOpt.cpp GetOpt.h
options.o: options.cpp stds.hpp GetOpt.h externs.hpp
DTree.o: DTree.cpp DTree.hpp stds.hpp BNode.hpp externs.hpp
BNode.o: BNode.cpp BNode.hpp stds.hpp externs.hpp
ParseData.o: ParseData.cpp DTree.hpp stds.hpp \
 BNode.hpp externs.hpp
ParseNames.o: ParseNames.cpp DTree.hpp stds.hpp \
 BNode.hpp externs.hpp
GetToken.o: GetToken.cpp stds.hpp
main.o: main.cpp stds.hpp externs.hpp GetOpt.h \
 BNode.hpp DTree.hpp
SplitID3.o: SplitID3.cpp stds.hpp externs.hpp \
 BNode.hpp DTree.hpp dVecLess.hpp
Train.o: Train.cpp DTree.hpp stds.hpp \
 BNode.hpp externs.hpp
PrintCTree.o: PrintCTree.cpp DTree.hpp stds.hpp \
 BNode.hpp externs.hpp
Pst.o: Pst.cpp DTree.hpp stds.hpp BNode.hpp \
 externs.hpp iti_pst.h
PrintRules.o: PrintRules.cpp DTree.hpp stds.hpp \
 BNode.hpp externs.hpp
maintest.o: maintest.cpp stds.hpp externs.hpp \
 GetOpt.h BNode.hpp DTree.hpp
mainlim.o: mainlim.cpp stds.hpp externs.hpp GetOpt.h \
 BNode.hpp DTree.hpp
```



```
/*
 * <ilic@hickory.engr.utk.edu>
 * $Log: GetOpt.h $
 * Revision 0.0 1993/01/24 22:04:56 ilic
 * This is GNU file. Observe the GNU copyright!
 *
*/

#ifndef _GetOpt_h
#define _GetOpt_h 1
#include <stdio.h>
#include <string.h>

// void *alloca(size_t);

#include <stdlib.h>

class GetOpt{
private:
  /* The next char to be scanned in the option-element
     in which the last option character we returned was found.
     This allows us to pick up the scan where we left off.

     If this is zero, or a null string, it means resume the scan
     by advancing to the next ARGV-element. */

  char *nextchar;


  /* Describe how to deal with options that follow non-option
ARGV-elements.

     UNSPECIFIED means the caller did not specify anything;
     the default is then REQUIRE_ORDER if the environment variable
     _OPTIONS_FIRST is defined, PERMUTE otherwise.

     REQUIRE_ORDER means don't recognize them as options.
     Stop option processing when the first non-option is seen.
     This is what Unix does.

     PERMUTE is the default. We permute the contents of 'argv' as we
scan,
     so that eventually all the options are at the end. This allows options
```

```
    to be given in any order, even with programs that were not written to
    expect this.

    RETURN_IN_ORDER is an option available to programs that were
written
    to expect options and other ARGV-elements in any order and that
care about
    the ordering of the two. We describe each non-option ARGV-element
    as if it were the argument of an option with character code zero.
    Using '-' as the first character of the list of option characters
    requests this mode of operation.

    The special argument '--' forces an end of option-scanning regardless
    of the value of 'ordering'. In the case of RETURN_IN_ORDER, only
    '--' can cause 'getopt' to return EOF with 'optind' != ARGC. */

  enum { REQUIRE_ORDER, PERMUTE, RETURN_IN_ORDER }
ordering;

  /* Handle permutation of arguments. */

  /* Describe the part of ARGV that contains non-options that have
     been skipped. 'first_nonopt' is the index in ARGV of the first of them;
     'last_nonopt' is the index after the last of them. */

  int first_nonopt;
  int last_nonopt;

  void exchange (char **argv);
public:
  /* For communication from 'getopt' to the caller.
     When 'getopt' finds an option that takes an argument,
     the argument value is returned here.
     Also, when 'ordering' is RETURN_IN_ORDER,
     each non-option ARGV-element is returned here. */

  char *optarg;

  /* Index in ARGV of the next element to be scanned.
     This is used for communication to and from the caller
     and for communication between successive calls to 'getopt'.
     On entry to 'getopt', zero means this is the first call; initialize.

     When 'getopt' returns EOF, this is the index of the first of the
     non-option elements that the caller should itself scan.

     Otherwise, 'optind' communicates from one call to the next
     how much of ARGV has been scanned so far. */

  int optind;

  /* Callers store zero here to inhibit the error message
     for unrecognized options. */

  int opterr;

  int nargc;
  char **nargv;
  char *noptstring;

  GetOpt (int argc, char **argv, char *optstring);
  int operator () (void);
};

#endif
// -*- C++ -*-
/* BNode.hpp - Jovan Ilic ilic@thetausa.com Mon Apr 24 2000
 *
 * $Header$
 * $Log$
 */

#ifndef _BNODE_HPP
#define _BNODE_HPP

#include "stds.hpp"

#include "externs.hpp"

class BNode {
private: // make clear that private part is first by default

  BNode *right;
  BNode *left;
  BNode *parent; // threaded bin trees
```

```
  bool pure, done;

  double threshold;
  int attribNo;
  // above values are internal representations.
  // label is not needed.

protected: // protected members


public: // public members

  vector<vector<double> * > data;

  BNode(); // default constructor

  BNode(const BNode &); // copy constructor

  BNode &operator=(const BNode &); // overloaded = operator

  ~BNode(); // virtual destructor (pure ??)

  BNode *Right()const ;
  void Right(const BNode *);

  BNode *Left() const ;
  void Left(const BNode *);

  BNode *Parent() const;
  void Parent(const BNode *);

  vector<vector<double> * > Data() const;
  void Data(const vector<vector<double> * > &dt);

  double Threshold() const;
  void Threshold(const double);

  int AttribNo() const;
  void AttribNo(const int);

  int Label() const;
  void Label(const int);

  bool Done() const;
  void Done(const bool);

  bool Pure() const;
  void Pure(const bool p);

  int label; // internal representation of a label


};

#endif
// -*- C++ -*-
/* DTree.hpp - Sun Sep 5 1999
 * <ilic@circpc15>
 *
 * $Header$
 * $Log$
 */

#ifndef _DTREE_HPP
#define _DTREE_HPP

#include "stds.hpp"

#include "BNode.hpp"

#include "externs.hpp"

// This is a base Decision Tree class. It provides minimal
// functionality and just serves to encapsulate the training
// set data, categories and methods unlikely to be changed in
// the derived classes. Derived classes should provide additional
// functionality for building the decision tree, different splitting
// criterias, impurity measures, complexity measures etc.

class DTree {
private: // make clear that private part is first by default

  enum {ERR = -1, ROOT, LEFT, RIGHT, CONT=5, DISC };

  int InitStats();
```

```
  string commentBeg;
  string delimStr;
  string termStr;
  string attrDelim;
  string dataDelim;
  string spaces;

  BNode *root, *curr;

  // classes map
  map<string, int, less<string> > class2int;
  map<int, string, less<int> > int2class;

  // Names do not have internal representation.
  vector<string> attribs;
  vector<int> types;

  // attr2int and int2attr are maps for valid attribute values.
  // if an attribute is continuous, its integer representation is -1.
  vector<map<string, int, less<string> > > attr2int;
  vector<map<int, string, less<int> > > int2attr;

  // all data is stored in double format and interpreted according to
  // attribute definition as continuous or categorical.
  // Categorical values are stored in internal format.
  vector<vector<double> > dset;

  // vector of vectors of leaf nodes to backtrack the rules
  // each vector of leaf nodes has the same label
  vector<vector<BNode *> > leaves;

  // critical variable
  double critVar;
  int critIndx;

protected: // protected members


public: // public members

  DTree(); // default constructor

  DTree(const DTree &); // copy constructor

  DTree &operator=(const DTree &); // overloaded = operator

  int ParseNames(string const &);
  int ParseData(string const &);
  int ParseData(string const &, vector<vector<double> > &);

  int PrintNames(ostream &, bool);
  int PrintData(ostream &, bool);

  virtual ~DTree(); // virtual destructor (pure ??)

  BNode *RootBNode() const {return root;}
  void RootBNode(const BNode *bn) { root = const_cast<BNode
*>(bn);}

  BNode *CurrBNode() const {return curr;}
  void CurrBNode(const BNode *cu){ curr = const_cast<BNode *>(cu);
}

  int Type(const int) const ;

  // This is a recursive function so it must have an argument
  int Height(BNode *t);

  // Non-recursive subtree traversal.
  void PrintCTree(ostream &, bool, BNode *t = NULL);

  // rule printing by backtracking from the leaves
  // if bool is true, print the internal format
  void PrintRules(ostream &, bool);

  // Train() is made generic by implementing different Split() functions
  // called by Train(). Split() could also be made a virtual function of
  // DTree class.
  BNode *Train();

  vector<map<string, int, less<string> > > &Attr2Int() {
    return attr2int;
  }

  vector<map<int, string, less<int> > > &Int2Attr() {
    return int2attr;
```

```
  }

  map<int, string, less<int> > &Int2Class() {
    return int2class;
  }

  int EvalInst(vector<double> const &);
  int TestInst(vector<double> const &);

  double TestTrainingSet();

  int SavePST(string const &);

  const vector<vector<double> > &DSet() { return dset; }

  int CritIndx() const { return critIndx;}
  double CritVar() const { return critVar;}

};

#endif
// -*- C++ -*-
/* dVecLess.hpp - Jovan Ilic ilic@thetausa.com Thu Jul 27 2000
 *
 * $Header$
 * $Log$
 */

#ifndef _DVECLESS_HPP
#define _DVECLESS_HPP

#include <vector>

// dVecLess is just a function object class providing comparison operator
// for sorting
class dVecLess {
private: // make clear that private part is first by default

  const int sortIndx;

protected: // protected members

public: // public members

  dVecLess() : sortIndx(0) {}; // default constructor

  dVecLess(const int indx) : sortIndx(indx) {};

  virtual ~dVecLess(){};

  bool operator()(std::vector<double> *p, std::vector<double> *q) {

    return ((*p)[sortIndx] < (*q)[sortIndx]);
  }

};


#endif
// -*- C++ -*-
/* externs.hpp - Sat Sep 4 1999
 * <ilic@circpc15>
 *
 * $Header$
 * $Log$
 */

#ifndef _EXTERNS_HPP
#define _EXTERNS_HPP

#define Log2 0.69314718055994530942
#define Log(x) ((x) <= 0 ? 0.0 : log(x) / Log2)

extern void usage(void);

extern char *fin, *fout, *cppout;
extern bool form;


void GetOptions(int, char **);
void GetOptionsHelp(int, char **);

string::size_type GetToken(string::size_type, string &, string &, string &);

#endif
```

```
// -*- C++ -*-
/* stds.hpp - Jovan Ilic ilic@thetausa.com Mon May 08 2000
 *
 * $Header$
 * $Log$
 */

#ifndef _STDS_HPP
#define _STDS_HPP

#include <iostream>
#include <string>
#include <stack>
#include <fstream>
#include <vector>
#include <map>
#include <algorithm>
#include <iterator>
#include <ctime>

#include <stdlib.h>

using std::vector;
using std::cout;
using std::cerr;
using std::endl;
using std::string;
using std::fstream;
using std::map;
using std::less;
using std::ostream;
using std::stack;
using std::ios;
using std::ostream_iterator;
using std::sort;
using std::clock;

#endif
// -*- C++ -*-
/* BNode.cpp - Jovan Ilic ilic@thetausa.com Tue Apr 25 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "BNode.hpp"


BNode::BNode(){ // default constructor

  // It will probably be needed for tree restructaring
  parent = NULL;
  right = NULL;
  left = NULL;
  threshold = 0.0;
  attribNo = -1; // invalid index
  label = -1; // invalid label map
  done = false;
  pure = false;
}

BNode::BNode(const BNode &bnd){ // copy constructor

  right = bnd.Right();
  left = bnd.Left();
  parent = bnd.Parent();
  data = bnd.Data();
  threshold = bnd.Threshold();
  attribNo = bnd.AttribNo();
  label = bnd.Label();
  done = bnd.Done();
  pure = bnd.Pure();
}

BNode &BNode::operator=(const BNode &bnd){ // overloaded =
operator

  right = bnd.Right();
  left = bnd.Left();
  parent = bnd.Parent();
  data = bnd.Data();
  threshold = bnd.Threshold();
  attribNo = bnd.AttribNo();
```

```
  label = bnd.Label();
  done = bnd.Done();
  pure = bnd.Pure();
  return *this;
}

BNode::~BNode(){};

BNode *BNode::Right() const { return right;}

void BNode::Right(const BNode *bnp) {
  right = const_cast<BNode *>(bnp);
  const_cast<BNode *>(bnp)->Parent(this);
}

BNode *BNode::Left() const { return left; }
void BNode::Left(const BNode *bnp) {
  left = const_cast<BNode *>(bnp);
  const_cast<BNode *>(bnp)->Parent(this);
}

BNode *BNode::Parent() const { return parent;}
void BNode::Parent(const BNode *bnp) {
  parent = const_cast<BNode *>(bnp);
}

vector<vector<double> *> BNode::Data() const { return data;}
void BNode::Data(const vector<vector<double> * > &dt) {
  data = dt;
}

double BNode::Threshold() const { return threshold; }
void BNode::Threshold(const double th) {
  threshold = th;
}

int BNode::AttribNo() const { return attribNo;}
void BNode::AttribNo(const int an) {
  attribNo = an;
}

int BNode::Label() const {
  return label;
}

}

void BNode::Label(const int lbl) {
  label = lbl;
}

bool BNode::Done() const { return done; }

void BNode::Done(const bool dn) {
  done = dn;
}

// determine if the samples belonging to a BNode
// have the same label
bool BNode::Pure() const {

  return pure;
}

void BNode::Pure(const bool p) {

  pure = p;
}
// -*- C++ -*-
/* DTree.cpp - Sun Sep 5 1999
 * <ilic@circpc15>
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "DTree.hpp"


int DTree::InitStats() {
  commentBeg = "|";
  delimStr = ",";
  termStr = ".";
  attrDelim = ":";
```

```
  dataDelim = ",";
  spaces = " \t\n\r";
  root = new BNode;
  curr = root;
  critVar = 0.0;
  critIndx = 0;

  return 0;

}

DTree::DTree() {
  InitStats();
}

DTree::DTree(DTree const &dt) {
  // this type of assignment will cause the
  // destructor to be called multiple times for the
  // same allocated memory for the binary tree
  // need reference counting or something similar
  // replicating the entire tree might be too memory/CPU
  // intensive
  root = dt.RootBNode();
  curr = dt.CurrBNode();
  InitStats();

}

DTree &DTree::operator=(DTree const &dt) {
  // this type of assignment will cause the
  // destructor to be called multiple times for the
  // same allocated memory for the binary tree
  // need reference counting or something similar
  // replicating the entire tree might be too memory/CPU
  // intensive
  InitStats();
  root = dt.RootBNode();
  curr = dt.CurrBNode();

  return *this;
}

// here we should clean up the BNode *root
// and vector<vector<double> *> dset;
DTree::~DTree() {

}

// Since this is a recursive function, it cannot have a
// default argument or anything like that.
int DTree::Height(BNode *tt) {
  BNode *t = tt;
  int h = 0;
  if(t != 0) {
    int lh = Height(t->Left());
    int rh = Height(t->Right());
    h = ((lh > rh) ? lh : rh) + 1;
  }
  return h;
}


int DTree::Type(const int indx) const {

  if((indx < 0) || (indx > types.size() - 1))
    return(ERR);
  else
    return(types[indx]);
}


// Evaluates an instance and returns the label
int DTree::EvalInst(vector<double> const &ins) {

  BNode *curr = RootBNode();
  bool branch;

  while(!curr->Pure()) {

    if(Type(curr->AttribNo()) == DISC)
      if(curr->Threshold() == ins[curr->AttribNo()])
  branch = true;
      else
  branch = false;

    if(Type(curr->AttribNo()) == CONT)
```

```
      if(ins[curr→AttribNo()] <= curr→Threshold())
  branch = true;
      else
  branch = false;

    if(branch)
      curr = curr→Left();
    else
      curr = curr→Right();
  }

  return(curr→Label());
}


double DTree::TestTrainingSet() {

  int i,k, right = 0, wrong = 0;
  int lblIndx = dset[0].size() - 2;

  for(i = 0; i < dset.size(); i++){
    if((k = EvalInst(dset[i])) == dset[i][lblIndx])
      right++;
    else
      wrong++;
  }

  // return the proportion of wrongly classified instances
  return(static_cast<double>(wrong)/(right+wrong));
}


int DTree::TestInst(const vector<double> &ins) {
  BNode *curr = RootBNode();
  bool branch;

  critVar = 1e10;

  while(!curr→Pure()) {

    if(Type(curr→AttribNo()) == DISC)
      if(curr→Threshold() == ins[curr→AttribNo()])
  branch = true;
      else
  branch = false;

    if(Type(curr→AttribNo()) == CONT){
      if(ins[curr→AttribNo()] <= curr→Threshold()){
  if(critVar > curr→Threshold() - ins[curr→AttribNo()]) {
    critVar = curr→Threshold() - ins[curr→AttribNo()];
    critIndx = curr→AttribNo();
  }
  branch = true;
      }else{
  if(critVar > ins[curr→AttribNo()] - curr→Threshold()) {
    critVar = ins[curr→AttribNo()] - curr→Threshold();
    critIndx = curr→AttribNo();
  }
  branch = false;
      }
    }

    if(branch)
      curr = curr→Left();
    else
      curr = curr→Right();
  }

  return(curr→Label());
}


// $Log: GetOpt.cpp $
// Revision 0.0 1993/01/15 19:19:51 ilic
// This is first and final version of GNU implementation
// of getopt function. Observe the GNU copyright.
//

#ifndef lint
static char *rcsid = "$Id:  getopt.cc 0.0 1993/01/15 19:19:51 ilic
Exp ilic $";
#endif

#include <stdlib.h>

#include "GetOpt.h"
```

```
GetOpt::GetOpt (int argc, char **argv, char *optstring): opterr (1)
{
  /* Initialize the internal data when the first call is made.
     Start processing options with ARGV-element 1 (since ARGV-element
0
     is the program name); the sequence of previously skipped
     non-option ARGV-elements is empty. */

  nargc = argc;
  nargv = argv;
  noptstring = optstring;
  first_nonopt = last_nonopt = optind = 1;
  optarg = nextchar = 0;

  /* Determine how to handle the ordering of options and nonoptions. */
  if (optstring[0] == '-')
    ordering = RETURN_IN_ORDER;
  else if (getenv ("_POSIX_OPTION_ORDER") != 0)
    ordering = REQUIRE_ORDER;
  else
    ordering = PERMUTE;
}

void
GetOpt::exchange (char **argv)
{
  int nonopts_size
    = (last_nonopt - first_nonopt) * sizeof (char *);
  char **temp = (char **) alloca (nonopts_size);

  /* Interchange the two blocks of data in argv. */
  memcpy (&argv[first_nonopt], temp, nonopts_size);
  memcpy (&argv[last_nonopt], &argv[first_nonopt],
    (optind - last_nonopt) * sizeof (char *));
  memcpy (temp, &argv[first_nonopt + optind - last_nonopt],
    nonopts_size);

  /* Update records for the slots the non-options now occupy. */
  first_nonopt += (optind - last_nonopt);
  last_nonopt = optind;
}
```



```
/* Scan elements of ARGV (whose length is ARGC) for option characters
   given in OPTSTRING.

   If an element of ARGV starts with '-', and is not exactly "-" or "--",
   then it is an option element. The characters of this element
   (aside from the initial '-') are option characters. If 'getopt'
   is called repeatedly, it returns successively each of theoption characters
   from each of the option elements.

   If 'getopt' finds another option character, it returns that character,
   updating 'optind' and 'nextchar' so that the next call to 'getopt' can
   resume the scan with the following option character or ARGV-element.

   If there are no more option characters, 'getopt' returns 'EOF'.
   Then 'optind' is the index in ARGV of the first ARGV-element
   that is not an option. (The ARGV-elements have been permuted
   so that those that are not options now come last.)

   OPTSTRING is a string containing the legitimate option characters.
   A colon in OPTSTRING means that the previous character is an option
   that wants an argument. The argument is taken from the rest of the
   current ARGV-element, or from the following ARGV-element,
   and returned in 'optarg'.

   If an option character is seen that is not listed in OPTSTRING,
   return '?' after printing an error message. If you set 'opterr' to
   zero, the error message is suppressed but we still return '?'.

   If a char in OPTSTRING is followed by a colon, that means it wants an
arg,
   so the following text in the same ARGV-element, or the text of the
following
   ARGV-element, is returned in 'optarg. Two colons mean an option that
   wants an optional arg; if there is text in the current ARGV-element,
   it is returned in 'optarg'.

   If OPTSTRING starts with '-', it requests a different method of
handling the
   non-option ARGV-elements. See the comments about
RETURN_IN_ORDER, above. */

int
GetOpt::operator () (void)
```

```
{
  if (nextchar == 0 || *nextchar == 0){
    if (ordering == PERMUTE){
      /* If we have just processed some options following some
non-options,
   exchange them so that the options come first. */

      if (first_nonopt != last_nonopt && last_nonopt != optind)
  exchange (nargv);
      else if (last_nonopt != optind)
  first_nonopt = optind;

      /* Now skip any additional non-options
   and extend the range of non-options previously skipped. */

      while (optind < nargc
       && (nargv[optind][0] != '-'
     || nargv[optind][1] == 0))
  optind++;
      last_nonopt = optind;
    }

    /* Special ARGV-element '-' means premature end of options.
       Skip it like a null option,
       then exchange with previous non-options as if it were an option,
       then skip everything else like a non-option. */

    if (optind != nargc && !strcmp (nargv[optind], "--")){
      optind++;

      if (first_nonopt != last_nonopt && last_nonopt != optind)
  exchange (nargv);
      else if (first_nonopt == last_nonopt)
  first_nonopt = optind;
      last_nonopt = nargc;

      optind = nargc;
    }

    /* If we have done all the ARGV-elements, stop the scan
       and back over any non-options that we skipped and permuted. */

    if (optind == nargc){
      /* Set the next-arg-index to point at the non-options
   that we previously skipped, so the caller will digest them. */
      if (first_nonopt != last_nonopt)
optind = first_nonopt;
      return EOF;
    }

    /* If we have come to a non-option and did not permute it,
       either stop the scan or describe it to the caller and pass it by. */

    if (nargv[optind][0] != '-' || nargv[optind][1] == 0){
      if (ordering == REQUIRE_ORDER)
return EOF;
      optarg = nargv[optind++];
      return 0;
    }

    /* We have found another option-ARGV-element.
       Start decoding its characters. */

    nextchar = nargv[optind] + 1;
}

/* Look at and handle the next option-character. */

{
  char c = *nextchar++;
  char *temp = strchr (noptstring, c);

  /* Increment 'optind' when we start to process its last character. */
  if (*nextchar == 0)
    optind++;

  if (temp == 0 || c == ':'){
    if (opterr != 0){
if (c < 040 || c >= 0177)
  fprintf (stderr, "%s:  unrecognized option, character code 0%o\n",
      nargv[0], c);
else
  fprintf (stderr, "%s:  unrecognized option '-%c'\n",
      nargv[0], c);
    }
    return '?';
```

```
    }
    if (temp[1] == ':'){
      if (temp[2] == ':'){
  /* This is an option that accepts an argument optionally. */
  if (*nextchar != 0){
    optarg = nextchar;
    optind++;
  }
  else
    optarg = 0;
  nextchar = 0;
      }else{
  /* This is an option that requires an argument. */
  if (*nextchar != 0){
    optarg = nextchar;
    /* If we end this ARGV-element by taking the rest as an arg,
       we must advance to the next element now. */
    optind++;
  }else
    if (optind == nargc){
      if (opterr != 0)
        fprintf (stderr, "%s:  no argument for '-%c' option\n",
            nargv[0], c);
      c = '?';
    } else
      /* We already incremented 'optind' once;
         increment it again when taking next ARGV-elt as argument. */
      optarg = nargv[optind++];
  nextchar = 0;
      }
    }
    return c;
  }
}
// -*- C++ -*-
/* GetToken.cpp - Jovan Ilic ilic@thetausa.com Tue May 2 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif
```

```
#include "stds.hpp"

// extract a token from a string starting at the position pos and
// return it in string token. Return value is the position of the next
// token or string::npos if it is the last token extracted.
// Leading and trailing white spaces are stripped off.
string::size_type GetToken(string::size_type pos, string &str,
          string &delimStr,
          string &token) {

  string spaces(" \t\n\r");

  string tmpStr;

  string::size_type posl, posr;

  // if the first character is a delimiter, jump over
  if(pos == str.find_first_of(delimStr, pos))
    posl = pos+1;
  else
    posl = pos;

  posr = str.find_first_of(delimStr, posl);


  token = str.substr(posl, posr - posl);
  token.erase(0, token.find_first_not_of(spaces));
  token.erase(token.find_last_not_of(spaces)+1, token.size());

  if(posr != string::npos) {
    return posr;
  } else {
    return string::npos;
  }



}
// -*- C++ -*-
/* main.cpp - Jovan Ilic ilic@thetausa.com Mon Apr 24 2000
 *
 * $Log$
```

```
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "stds.hpp"

#define extern
#include "externs.hpp"
#undef extern

#include "GetOpt.h"

#include "BNode.hpp"
#include "DTree.hpp"

int main(int argc, char **argv)
{

  fin = "none";
  fout = "none";
  cppout = "none";
  form = false;

  form = false;

  GetOptions(argc, argv);

  string InputFile(fin);
  if(InputFile == "none"){
    GetOptionsHelp(argc, argv);
    exit(EXIT_FAILURE);
  }

  DTree test;

  // all the input files have the same stem and appropriate suffixes
  string NamesFile = InputFile + ".names";
  string DataFile = InputFile + ".data";
  string AllDataFile = InputFile + ".all";
  string TestFile = InputFile + ".test";

  test.ParseNames(NamesFile);
  //test.PrintNames(cout, form);
  test.ParseData(DataFile);
  //test.PrintData(cout, form);

  test.Train();

  // save the trained tree in internal format
  string TreeFile(fout);
  if(TreeFile != "none") {

  }

  string CPPFile(cppout);
  if(CPPFile != "none") {
    fstream cppos(CPPFile.c_str(), ios::out);
    test.PrintCTree(cppos, true);
    cppos.close();

  }

  double tst = test.TestTrainingSet();
  cout << "missclassified training set instances = " << tst << "%" <<
endl;
  cout << "Height of the tree is " << test.Height(test.RootBNode()) <<
endl;

  return 0;
}

// -*- C++ -*-
/* main.cpp - Jovan Ilic ilic@thetausa.com Mon Apr 24 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "stds.hpp"
```

```
#define extern
#include "externs.hpp"
#undef extern

#include "GetOpt.h"

#include "BNode.hpp"
#include "DTree.hpp"

int main(int argc, char **argv)
{

  fin = "none";
  fout = "none";
  cppout = "none";
  form = false;

  form = false;

  GetOptions(argc, argv);

  string InputFile(fin);
  if(InputFile == "none"){
    GetOptionsHelp(argc, argv);
    exit(EXIT_FAILURE);
  }

  DTree test;

  // all the input files have the same stem and appropriate suffixes
  string NamesFile = InputFile + ".names";
  string DataFile = InputFile + ".data";
  string AllDataFile = InputFile + ".all";
  string TestFile = InputFile + ".test";


  test.ParseNames(NamesFile);
  //test.PrintNames(cout, form);
  test.ParseData(DataFile);
  //test.PrintData(cout, form);

  test.Train();

  // save the trained tree in internal format
  string TreeFile(fout);
  if(TreeFile != "none") {

  }

  string CPPFile(cppout);
  if(CPPFile != "none") {
    fstream cppos(CPPFile.c_str(), ios::out);
    test.PrintCTree(cppos, true);
    cppos.close();

  }

  vector<vector<double> > insts;

  test.ParseData(argv[3], insts);

  int lblIndx = insts[0].size() - 2;
  int wright = 0, wrong = 0;

  for(int i = 0; i < insts.size(); i++){

    if(insts[i][lblIndx] == test.TestInst(insts[i])){
      wright++;
    } else {
      wrong++;
      for(int j = 0; j < insts[0].size() - 1; j++)
cout << insts[i][j] << '\t';
      cout << "C[" << test.CritIndx() << "] = " << test.CritVar() << endl;

    }
  }

  cout << "Missclasified test rate = " <<
static_cast<double>(wrong)/static_cast<double>(wright+wrong)*100.0
<< "%" << endl;

  return 0;
}

// -*- C++ -*-
/* main.cpp - Jovan Ilic ilic@thetausa.com Mon Apr 24 2000
```

```
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "stds.hpp"

#define extern
#include "externs.hpp"
#undef extern

#include "GetOpt.h"

#include "BNode.hpp"
#include "DTree.hpp"

extern int dTree(const vector<double> &);

int main(int argc, char **argv)
{

  fin = "none";
  fout = "none";
  cppout = "none";
  form = false;

  form = false;

  GetOptions(argc, argv);

  string InputFile(fin);
  if(InputFile == "none"){
    GetOptionsHelp(argc, argv);
    exit(EXIT_FAILURE);
  }

  DTree test;

  // all the input files have the same stem and appropriate suffixes
  string NamesFile = InputFile + ".names";
  string DataFile = InputFile + ".data";
  string AllDataFile = InputFile + ".all";
  string TestFile = InputFile + ".test";


  test.ParseNames(NamesFile);
  //test.PrintNames(cout, form);
  test.ParseData(DataFile);
  //test.PrintData(cout, form);

  const vector<vector<double> > &ds = test.DSet();

  /*
  for(int i = 0; i < ds.size(); i++) {
    copy(ds[i].begin(), ds[i].end(), ostream_iterator<double>(cout, "\t"));
    cout << endl;
  }
  */

  int lblIndx = ds[0].size() - 2, cls, wright = 0, wrong = 0;

  for(int i = 0; i < ds.size(); i++){
    cls = dTree(ds[i]);
    if(cls != ds[i][lblIndx])
      wrong++;
    else
      wright++;
  }

  cout << "Misclassified " <<
(static_cast<double>(wrong)/static_cast<double>(wrong + wright) *
100.0);
  cout << "% test samples" << endl;

  return 0;
}

/* - J.Ilic e:/ilic/nn/pnn/specht/options.cc
 * Wed Oct 14 1992
 * <ilic@morse.eece.maine.edu>
 */

// $Log$
```

```
#ifndef lint
static char *rcsid = "$Id";
#endif

#include "stds.hpp"

#include "GetOpt.h"

#include "externs.hpp"


// The following functions prints out help (option -?) and/or warning
// messages
void GetOptionsHelp(int argc, char **argv)
{
  cerr << endl;
  cerr << "Usage:   " << endl;
  cerr << argv[0] << " -f <input file> -o <output file> -?";
  cerr << endl;
  cerr << "Where a flag can be either small or capital letter and
the meaning is:  " << endl;
  cerr << '\t' << "-i <input file> is the input file." << endl;
  cerr << '\t' << "-o <dt file> is the decision tree output file." <<
endl;
  cerr << '\t' << "-c <c++ file> is the C++ tree representation."
<< endl;
  cerr << '\t' << "-f print internal data representation." << endl;
  cerr << '\t' << "-?  Prints this message." << endl;
};

// GetOptions() parses the command line and initializes the
// data structures needed
void GetOptions(int argc, char **argv)
{

  GetOpt getopt(argc, argv, "C:c:I:i:Ff0:o:h:?");
  int option_char;
  while((option_char = getopt()) != EOF)
    switch(option_char){
      // domain vectors file
    case 'I':
    case 'i':
      fin = getopt.optarg;
      break ;
      // range vectros file
      break;
      // output file
    case 'O':
    case 'o':
      fout = getopt.optarg;
      break;
    case 'c':
    case 'C':
      cppout = getopt.optarg;
      break;
      // output format: internal = i, external = o
    case 'f':
      form = true;
      break;
    case '?':
    case 'h':
      GetOptionsHelp(argc, argv);
      exit(EXIT_FAILURE);
    }

};
// -*- C++ -*-
/* ParseData.cpp - Wed Sep 22 1999
 * <ilic@circpc15>
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "DTree.hpp"

int DTree::ParseData(string const &insts)
{

  const int LN_SIZE = 1024;

  int dcount, icount;
```

```
char ln[LN_SIZE];

string lnStr, tkn;
lnStr.reserve(LN_SIZE);
tkn.reserve(LN_SIZE);

string::size_type tmpPos;

fstream dt(insts.c_str(), ios::in);

// should use exceptions here!!
if(!dt) {
  cerr << "File:  " << __FILE__ << " line:  " << __LINE__ << endl;
  cerr << "Could not open:  " << insts << " for reading." << endl;
  exit(-1);
}

icount = 0;

while(dt.getline(ln, LN_SIZE-1)) {

  lnStr = ln;

  // ignore empty lines
  if(lnStr.find_first_not_of(spaces) == string::npos)
    continue;

  // strip off the lleading and trailing white spaces
  lnStr.erase(0, lnStr.find_first_not_of(spaces));
  lnStr.erase(lnStr.find_last_not_of(spaces)+1, lnStr.length());

  // check for whole line comments
  if(lnStr[0] == '|') {
    continue;
  }

  // strip off data terminator (.)
  tmpPos=lnStr.find_last_of(termStr);
  lnStr.erase(tmpPos, lnStr.length()-tmpPos);

  tmpPos = 0;
  dset.push_back(vector<double>());
    dcount = 0;

    while((tmpPos=GetToken(tmpPos, lnStr, delimStr, tkn)) !=
string::npos) {

      if(types[dcount] == CONT) // continuous
  dset[icount].push_back(atof(tkn.c_str()));
      else // discrete
  dset[icount].push_back(attr2int[dcount][tkn]);
      dcount++;
    }

    dset[icount].push_back(class2int[tkn]);

    // instance index needs to go with the instances
    // to keep maps straight, it is pushed at the end
    dset[icount].push_back(icount);

    icount++;
  }

  return 0;
}


int DTree::ParseData(string const &insts, vector<vector<double> >
&tst)
{

  const int LN_SIZE = 1024;

  int dcount, icount;

  char ln[LN_SIZE];

  string lnStr, tkn;
  lnStr.reserve(LN_SIZE);
  tkn.reserve(LN_SIZE);

  string::size_type tmpPos;

  fstream dt(insts.c_str(), ios::in);
```

```
  // should use exceptions here!!
  if(!dt) {
    cerr << "File:  " << __FILE__ << " line:  " << __LINE__ << endl;
    cerr << "Could not open:  " << insts << " for reading." << endl;
    exit(-1);
  }

  icount = 0;

  while(dt.getline(ln, LN_SIZE-1)) {

    lnStr = ln;

    // ignore empty lines
    if(lnStr.find_first_not_of(spaces) == string::npos)
      continue;

    // strip off the lleading and trailing white spaces
    lnStr.erase(0, lnStr.find_first_not_of(spaces));
    lnStr.erase(lnStr.find_last_not_of(spaces)+1, lnStr.length());

    // check for whole line comments
    if(lnStr[0] == '|') {
      continue;
    }

    // strip off data terminator (.)
    tmpPos=lnStr.find_last_of(termStr);
    lnStr.erase(tmpPos, lnStr.length()-tmpPos);

    tmpPos = 0;
    tst.push_back(vector<double>());
    dcount = 0;

    while((tmpPos=GetToken(tmpPos, lnStr, delimStr, tkn)) !=
string::npos) {

      if(types[dcount] == CONT) // continuous
  tst[icount].push_back(atof(tkn.c_str()));
      else // discrete
  tst[icount].push_back(attr2int[dcount][tkn]);
      dcount++;
    }

    tst[icount].push_back(class2int[tkn]);

    // instance index needs to go with the instances
    // to keep maps straight, it is pushed at the end
    tst[icount].push_back(icount);

    icount++;
  }

  return 0;
}



int DTree::PrintData(ostream &os, bool internal)
{

  int i, j;


  // the last column is the instance index
  for(i = 0; i < dset.size(); i++){

#ifdef DEBUG
    os << dset[i][dset[i].size() - 1] << ":   ";
#endif

    for(j = 0; j < dset[i].size() - 1; j++)
      if(j < (dset[i].size()-2))
  if(internal)
    os << dset[i][j] << ',';
  else
    if(types[j] != CONT) // discrete
      os << int2attr[j][dset[i][j]] << ',';
    else
      os << dset[i][j] << ',';

      else
  if(internal)
    os << dset[i][j] << '.' << endl;
  else
    os << int2class[dset[i][j]] << '.' << endl;
```

```
  }

  return 0;
}
// -*- C++ -*-
/* ParseNames.cpp - Sun Sep 5 1999
 * <ilic@circpc15>
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "DTree.hpp"

// labels are on the single line as well as the
// variable definitions.
int DTree::ParseNames(string const &names)
{

  const int LN_SIZE = 1024;

  bool CLASS = true;
  char ln[LN_SIZE];

  int classLbl = 0, atCnt = 0;

  vector<int> attrCnt;

  string::size_type tmpPos;

  string lnStr, tkn;
  lnStr.reserve(LN_SIZE);
  tkn.reserve(LN_SIZE);

  fstream nm(names.c_str(), ios::in);


  // should use exceptions here!!
  if(!nm) {
    cerr << "File:  " << __FILE__ << " line:  " << __LINE__ << endl;
    cerr << "Could not open:  " << names << " for reading." << endl;
    exit(-1);
  }

  while(nm.getline(ln, LN_SIZE-1)) {

    lnStr = ln;

    // ignore empty lines
    if(lnStr.find_first_not_of(spaces) == string::npos)
      continue;

    // strip off the lleading and trailing white spaces
    lnStr.erase(0, lnStr.find_first_not_of(spaces));
    lnStr.erase(lnStr.find_last_not_of(spaces)+1, lnStr.length());


    // check for whole line comments
    if(lnStr[0] == '|') {
      continue;
    }

    // strip off data terminator (.)
    tmpPos=lnStr.find_last_of(termStr);
    lnStr.erase(tmpPos, lnStr.length()-tmpPos);

    // reading classes labels
    if(CLASS) {

      tmpPos = GetToken(0, lnStr, delimStr, tkn);

      while(tmpPos != string::npos) {
class2int[tkn] = classLbl;
int2class[classLbl] = tkn;
classLbl++;

tmpPos = GetToken(tmpPos, lnStr, delimStr, tkn);
      }

      class2int[tkn] = classLbl;
      int2class[classLbl] = tkn;

      CLASS = false;
```

```
  } else { // class description are scaned.

    // get the attribut name
    tmpPos = GetToken(0, lnStr, attrDelim, tkn);
    attribs.push_back(tkn);

    attrCnt.push_back(0);
    types.push_back(DISC);

    // index offset of 1 is jump over ':' which is not the delimiter
    // anymore.
    tmpPos = GetToken(tmpPos+1, lnStr, delimStr, tkn);

    attr2int.push_back(map<string, int, less<string> >());
    int2attr.push_back(map<int, string, less<int> >());

    if(tkn.find("continuous") != string::npos){
attrCnt[atCnt] = -1;
attr2int[atCnt][tkn] = attrCnt[atCnt];
int2attr[atCnt][attrCnt[atCnt]] = tkn;
types[atCnt] = CONT;
atCnt++;
continue;
    }

    while(tmpPos != string::npos) {

attr2int[atCnt][tkn] = attrCnt[atCnt];
int2attr[atCnt][attrCnt[atCnt]] = tkn;
attrCnt[atCnt]++;

tmpPos = GetToken(tmpPos, lnStr, delimStr, tkn);

    }

    attr2int[atCnt][tkn] = attrCnt[atCnt];
    int2attr[atCnt][attrCnt[atCnt]] = tkn;

    atCnt++;
  }

}
```

```
  return 0;

}


int DTree::PrintNames(ostream &os, bool internal)
{

  int i;

  std::map<int, string, less<int> >::iterator i2sp;

  os << "| Classses :  " << endl << endl;

  for(i2sp = int2class.begin(); i2sp != int2class.end(); ){
    if(internal)
      os << i2sp->first;
    else
      os << i2sp->second;
    if(++i2sp == int2class.end())
      os << '.' << endl << endl;
    else
      os << ',';
  }

  os << "| Attributes with possible values :  " << endl;

  for(i = 0; i < attribs.size(); i++) {

    os << attribs[i] << ":\t";

    for(i2sp = int2attr[i].begin(); i2sp != int2attr[i].end();){
      if(internal)
os << i2sp->first;
      else
os << i2sp->second;
      if(++i2sp == int2attr[i].end())
os << '.' << endl;
      else
os << ',';
    }
  }
```

```
  return 0;
}


ostream &operator≪(ostream &os, DTree const &dt)
{
  return os;
}

// -*- C++ -*-
/* PrintCTree.cpp - Jovan Ilic ilic@thetausa.com Tue Aug 08 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "DTree.hpp"

// Non-recursive subtree traversal.
// This is just an example of how to print a tree. Remeber that there are
// internal and external data representation. Generating code is probably
// too complicated due to this difference but it should be easy to generate
// various graphical/text representations of the tree as well as statistics.
void DTree::PrintCTree(ostream &os, bool internal, BNode *t) {

  time_t timer;
  time(&timer);

  if(t == NULL)
    t = RootBNode();

  int i;
  stack<BNode *> path;

  path.push(t);
  bool lft = true;

  // print out the header
  os ≪ "\n/*\n\tThis file was produced by:  " ≪ "dt" ≪ endl;
  os ≪ "\tThe current date and time is:  " ≪ ctime(&timer);
  os ≪ "\tThe training file is from the set:  " ≪ fin ≪ "\n*/" ≪
endl ≪ endl;

  os ≪ "#include <vector>" ≪ endl;
  os ≪ "using namespace std;" ≪ endl ≪ endl;
  os ≪ "int dTree(vector<double> const &C) {" ≪ endl;

  while(!path.empty()) {
    t = path.top(); path.pop();
    if(t→Pure()) {
      for(i = 0; i < path.size(); i++)
  os ≪ ' ';
      if(lft)
  os ≪ " return(" ≪ t→Label() ≪ ");" ≪ endl;
      else {
  os ≪ "else" ≪ endl;
  for(i = 0; i < path.size() + 1; i++)
    os ≪ ' ';
  os ≪ "return(" ≪ t→Label() ≪ ");" ≪ endl;
      }
      lft = false;
    } else {
      for(i = 0; i < path.size(); i++)
  os ≪ ' ';
      if(lft) {
  if(Type(t→AttribNo()) == DISC)
    os ≪ "if" ≪ "(C[" ≪ t→AttribNo() ≪ "] == " ≪ t→Threshold()
≪ ")" ≪ endl;
  if(Type(t→AttribNo()) == CONT)
    os ≪ "if" ≪ "(C[" ≪ t→AttribNo() ≪ "] <= " ≪ t→Threshold()
≪ ")" ≪ endl;
      } else {
  os ≪ "else" ≪ endl;;
  for(i = 0; i < path.size() + 1; i++)
    os ≪ ' ';
  if(Type(t→AttribNo()) == DISC)
    os ≪ "if" ≪ "(C[" ≪ t→AttribNo() ≪ "] == " ≪ t→Threshold()
≪ ")" ≪ endl;
  if(Type(t→AttribNo()) == CONT)
    os ≪ "if" ≪ "(C[" ≪ t→AttribNo() ≪ "] <= " ≪ t→Threshold()
≪ ")" ≪ endl;
  lft = true;
```

```
      }

      path.push(t→Right());
      path.push(t→Left());
      lft = true;
    }
  }
  os ≪ "} // End of dTree() " ≪ endl;
}

// -*- C++ -*-
/* PrintRules.cpp - Jovan Ilic ilic@thetausa.com Fri Aug 11 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "DTree.hpp"
#include "stds.hpp"

void DTree::PrintRules(ostream &os, bool form) {

  int i, j;
  BNode *nd;

  stack<BNode *> lStack;

  for(i = 0; i < leaves.size(); i++) {
    for(j = 0; j < leaves[i].size(); j++){
      nd = leaves[i][j];
      // empty the stack first
      while(!lStack.empty())
lStack.pop();
      do{
lStack.push(nd);
nd = nd→Parent();
      } while(nd != NULL);

      while(!lStack.empty()){
nd = lStack.top(); lStack.pop();
      }
    }
  }
}

// -*- C++ -*-
/* Pst.cpp - Jovan Ilic ilic@thetausa.com Fri Aug 4 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "DTree.hpp"
#include "iti_pst.h"

int DTree::SavePST(string const &pstf) {

  fstream pststr(pstf.c_str(), ios::out);

  if(!pststr) {
    cerr ≪ "Could not open for reading:  " ≪ pstf ≪ endl;
    exit(-1);
  }

  BNode *tree_node = root;

  return 0;
}

// -*- C++ -*-
/* SplitID3.cpp - Jovan Ilic ilic@thetausa.com Tue Jul 18 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif
```

```
#include <math.h>

#include "stds.hpp"
#include "externs.hpp"
#include "BNode.hpp"
#include "DTree.hpp"

#include "dVecLess.hpp" // function object

double Entropy(const vector<int> &);

// this is for numerical vectors
template<class T>
T Sum(const vector<T> &vec) {

  T sum = 0;
  for(int i = 0; i < vec.size(); i++)
    sum += vec[i];

  return sum;
}

clock_t total = 0;


// SplitID3 should always be passed a node that needs to be
// split, so there are certainly subbranches. dt is needed for
// additional information about the instances, etc.
bool SplitID3(BNode *nd, DTree &dt) {

  time_t sortTime;

  // this enum should match the enum in DTree.hpp
  enum {ERR = -1, ROOT, LEFT, RIGHT, CONT=5, DISC };

  int i, j, k; // counter declarations to make MSC happy
  double E = 0.0, bestGain = 0.0; // overall entropy and info gain


  // gain calculations
  int yesSum, noSum, sum;
  double yesE,noE,subG,Gain = 0.0;

vector<vector<double> *> &insts = nd->data;

double featureSplit; // continuous or descrete splitting value
int featureIndx; // feature variable index used for splitting

// children nodes
BNode *leftBr = new BNode();
BNode *rightBr = new BNode();

// label counts used for overall entropy and gain calculations
vector<int> lblCount(dt.Int2Class().size(), 0);

// the last column is instance index
int lblIndx = insts[0]->size() - 2;

// counts needed for information gain calculations
vector<int> yesCount(dt.Int2Class().size(),0);
vector<int> noCount(dt.Int2Class().size(),0);


// everything needed to find the overall entropy
for(i = 0; i < insts.size(); i++){
  lblCount[(*insts[i])[lblIndx]]++;
}

// max(Entropy) should be less than log2(number_of_labels)
E = Entropy(lblCount);

for(i = 0; i < insts[0]->size() - 1; i++) {

  if(dt.Type(i) == CONT) {

    sortTime = clock();
    sort(insts.begin(), insts.end(), dVecLess(i));
    total += clock() - sortTime;

   // vector of possible thresholds. There should already be
   // more than one label for each variable.
   vector<double> threshs;
   for(j = 0; j < insts.size()-1; j++)
     if( (*insts[j])[lblIndx] != (*insts[j+1])[lblIndx] ){
// do not duplicate thresholds
```

```
double threshsTmp = ((*insts[j])[i] + (*insts[j+1])[i] )/2.0;
if(threshs.size() == 0) {
  threshs.push_back(threshsTmp);
  continue;
}
if (threshsTmp != threshs[threshs.size() - 1])
  threshs.push_back( threshsTmp);
    }


  // counting could be optimized a little bit more
  for(j = 0; j < threshs.size(); j++){
    for(k = 0; k < yesCount.size(); k++) {
yesCount[k] = 0;
noCount[k] = lblCount[k];
    }

    // equality because the threshold could be the same as
    // feature variable value
    for(k=0; k < insts.size(); k++)
if((*insts[k])[i] <= threshs[j]){
  yesCount[(*insts[k])[lblIndx]]++;
  noCount[(*insts[k])[lblIndx]]--;
}


    yesSum = Sum(yesCount);
    noSum = Sum(noCount);
    sum = yesSum + noSum;

    yesE = Entropy(yesCount);
    noE = Entropy(noCount);


    subG =
(static_cast<double>(yesSum)/static_cast<double>(sum)*yesE + \
   static_cast<double>(noSum)/static_cast<double>(sum)*noE);

    Gain = E - subG;

    // record best Gain, best threshold, best spliting variable index
    if(bestGain < Gain){
  bestGain = Gain;
  featureSplit = threshs[j];
  featureIndx = i;
  // purety can be defined in different ways
  // node does not have to be really "pure" to be defined as such
  if(yesE == 0.0)
    leftBr->Pure(true);
  else
    leftBr->Pure(false);
  if(noE == 0.0)
    rightBr->Pure(true);
  else
    rightBr->Pure(false);

      }
    }
  } // if(dt.Type(i) == CONT)

  // now do the same if the variable is discrete

  if(dt.Type(i) == DISC) {
    // here we do not need any sorting or threshold
    // finding, only yes-no counting.

    int noFeatures = dt.Attr2Int()[i].size();

    // for each feature value
    for(k = 0; k < noFeatures; k++) {

// initialize the counters
for(j = 0; j < yesCount.size(); j++){
  yesCount[j] = 0;
  noCount[j] = lblCount[j];
}

// count instances with yes-no decisions
for(j = 0; j < insts.size(); j++){
  if((*insts[j])[i] == k){
    yesCount[(*insts[j])[lblIndx]]++;
    noCount[(*insts[j])[lblIndx]]--;
  }
}

// calculate information gain
```

```
  yesSum = Sum(yesCount);
  noSum = Sum(noCount);
  sum = yesSum + noSum;

  yesE = Entropy(yesCount);
  noE = Entropy(noCount);

  subG =
(static_cast<double>(yesSum)/static_cast<double>(sum)*yesE +\
    static_cast<double>(noSum)/static_cast<double>(sum)*noE);

  Gain = E - subG;
  // record bestGain, best decision variable and value
  if(bestGain < Gain) {
    bestGain = Gain;
    featureSplit = k;
    featureIndx = i;
    // purety can be defined in different ways
    // node does not have to be really "pure" to be defined as such
    if(yesE == 0.0)
      leftBr→Pure(true);
    else
      leftBr→Pure(false);
    if(noE == 0.0)
      rightBr→Pure(true);
    else
      rightBr→Pure(false);

  }
      }
    }
  }

  // from the splitting information, form the children nodes and connect
them
  // to the parent node that is being split.

  // store splitting info at node.
  nd→AttribNo(featureIndx);
  nd→Threshold(featureSplit);

  // connect the parent node
  leftBr→Parent(nd); rightBr→Parent(nd);

  // connect the children nodes
  nd→Right(rightBr); nd→Left(leftBr);

  // separate pointers to sample instances
  for(i = 0; i < insts.size(); i++){
    if(dt.Type(featureIndx) == DISC)
      if((*insts[i])[featureIndx] == featureSplit)
  leftBr→data.push_back(insts[i]);
      else
  rightBr→data.push_back(insts[i]);

    if(dt.Type(featureIndx) == CONT)
      if((*insts[i])[featureIndx] <= featureSplit)
  leftBr→data.push_back(insts[i]);
      else
  rightBr→data.push_back(insts[i]);
  }

  // if childern are pure assign them labels
  if(leftBr→Pure()){
    leftBr→Label((*leftBr→data[0])[lblIndx]);
  }
  else
    leftBr→Label(-1);

  if(rightBr→Pure()){
    rightBr→Label((*rightBr→data[0])[lblIndx]);
  }
  else
    rightBr→Label(-1);

  return true;
}


double Entropy(const vector<int> &data) {

  int i;
  double S = 0.0, e = 0.0;

  for(i = 0; i < data.size(); i++)
    S += data[i];
```

```
  for(i = 0; i < data.size(); i++) {
      e -=
(static_cast<double>(data[i])/S)*(Log(static_cast<double>(data[i])/S));
  }
  return e;
}
// -*- C++ -*-
/* Train.cpp - Jovan Ilic ilic@thetausa.com Fri Jun 02 2000
 *
 * $Log$
 */

#ifndef lint
static char *rcsid = "$Header$";
#endif

#include "DTree.hpp"

#include "externs.hpp"

bool SplitID3(BNode *, DTree &);

BNode *DTree::Train() {

  int i, lblIndx, lbl;

  stack<BNode *> nodes;

  curr = root = new BNode();

  // Initialize the root node.
  curr->data.reserve(dset.size());

  for(i = 0; i != dset.size(); i++)
    curr->data.push_back(&dset[i]);

  // root node contains no instances
  // children pointers are NULL
  if(curr->data.size() < 1 ){
    curr->Label(ERR);
    return curr;
  }

  lblIndx = curr->data[0]->size() - 2;

  // root node is pure
  // children pointers are NULL
  if(curr->Pure()){
    curr->Label((*curr->data[0])[lblIndx]);
    return curr;
  }

  // push root to be revisited later on
  nodes.push(root);

  leaves.erase(leaves.begin(), leaves.end());

  for(i = 0; i < int2class.size(); i++)
    leaves.push_back(vector<BNode *>());

  // root node has instances not belonging to the same class
  // we'll split them by minimizing the entropy
  while(!nodes.empty()) { // while there are nodes to split

    // get the next node from the stack
    curr = nodes.top();
    nodes.pop();

    // this seems to be the only ID3 specific point in TrainID3 function
    // SplitID3 splits current node using ID3 entropy minimization
    // and assigns the children nodes.
    SplitID3(curr, *this);

    // Here we assign the node label if it is a pure node. Adding new
    // instances to the node must invalidate the label if the purity is not
preserved.
    if(!curr->Right()->Pure()){
      nodes.push(curr->Right());
    }
    else {
      lbl = (*curr->Right()->data[0])[lblIndx];
      curr->Right()->Label(lbl);
      leaves[lbl].push_back(curr->Right());
    }

    // The same is done with the left child. Note that we always
```

```
// follow the left branch first.
if(!curr→Left()→Pure()){
  nodes.push(curr→Left());
}
else {
  lbl = (*curr→Left()→data[0])[lblIndx];
  curr→Left()→Label(lbl);
  leaves[lbl].push_back(curr→Left());
}
```

```
  }

  extern clock_t total;

  cout << "Sorting time = " << total << endl;

  return 0;

}
```

Vita

Jovan Ilić was born in Zaječar, Yugoslavia in 1962 where he lived for 18 years. From 1985 until 1989 he attended the University of Illinois Urbana-Champaign, where he received his Bachelor of Science Degree in Electrical and Computer Engineering. He began his graduate studies at the University of Tennessee, Knoxville in the Spring of 1990. After receiving his Master of Science degree in Electrical and Computer Engineering in 1992, he spent a year at The University of Maine. In 1994 he returned to The University of Tennessee to continue his graduate studies. Between working full time in the computer industry for a couple of years, and as an independent computer consultant for three years, he completed his Doctor of Philosophy studies in Spring of 2001.