NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

SOLUTION OF OPTIMIZING AND FORECASTING PROBLEMS BY USING GENETIC ALGORITHMS

Graduation Project
COM- 400

STUDENT: Naveed Mustafa

SUPERVISOR: Asst.Prof.Dr Rahib Abiyev

Nicosia-2001

ACKNOWLEDGMENTS

First of all I am happy that Allah Taa'la The Almighty Supreme Being and Hazrat Muhammad (peace be upon him) has provided me with the strength and courage to complete the task .

I wish to thank my advisor. Assistant Professor Dr. Rahib Abiyev for intellectual support, encouragement, and enthusiasm, which made this project possible, and his patience for correcting both my stylistic and scientific errors. I wish him success in his future life.

I want to dedicate my graduation project to my family. I am very thankful to my father Mr.Ghulam Mustafa who has financed my education and I want to thank my mother Mrs.Ghulam Mustafa for their support and prays for my success. I am proud of my brother Mr.Rafaq Mustafa how was always there to help me and finally my sisters Ms.Shagufta & Ms.Siqa the only thing I missed during my stay in university. I have an endless love for my family.

My sincerest thanks must go to my friends especially to Mr. Jamal Ashiq Qureshi , Mr. Ammar Ali, Mr. Babar Rehman, Mr. Hafiz Zulfiqar Ali, Mr. Awais Janjua and Mr.Faisal Mir who shared their suggestions and evaluations throughout the completion of the project and in my graduation. It is true that good friends are blessing.

ABSTRACT

Due to the complexity of the processes, it has become very difficult to control them on the base of traditional methods. In such condition it is necessary to use modern methods for solving these problems. One of such method is global optimization algorithm based on mechanics of natural selection and natural genetics, which is called Genetic Algorithms. In this project the application problems of genetic algorithms for optimization problems, its specific characters and structures are given. The Basic Genetic operations, Selection, Reproduction, Crossover, Encoding and Mutation operations are widely described. The affectivity of genetic algorithms for solving the genetic algorithms is shown in the following chapters. After the representation of optimization problems, structural optimization and the finding of optimal solution of quadratic equation are given.

The practical application for selection, reproduction, Crossover and mutation operation are shown. The functional implementation of GA based optimization in MATLAB Toolbox is considered.

TABLE OF CONTENTS

INTRODUCTION

The GENETIC ALGORITHMS is a model of machine learning, which derives its behavior from a metaphor of the processes of EVOLUTION in nature. This is done by the creation within a machine of a POPULATION of INDIVIDUALS represented by CHROMOSOMES, in essence of a set of character strings that are analogous to the base-4 chromosomes that we see in our own DNA. The individuals in the population then go through a process of evolution.

Genetic algorithms (GA) seek to solve optimization problems using the methods of evolution, specifically survival of the fittest. In a typical optimization problem, there are a number of variables, which control the process, and a formula or algorithm, which combines the variables to fully model the process. The problem is then to find the values of the variables, which optimize the model in some way. If the model is a formula, then we will usually be seeking the maximum or minimum value of the formula. There are many mathematical methods, which can optimize problems of this nature (and very quickly) for fairly "well-behaved" problems. These traditional methods tend to break down when the problem is not so "well-behaved." We should note that EVOLUTION (in nature or anywhere else) is not a purposive or directed process. That is, there is no evidence to support the assertion that the goal of evolution is to produce Mankind. Indeed, the processes of nature seem to boil down to different Individuals competing for resources in the ENVIRONMENT. Some are better than others, those that are better are more likely to survive and propagate their genetic material. In nature, we see that the encoding for our genetic information (GENOME) is done in a way that admits asexual REPRODUCTION (such as by budding) typically results in OFFSPRING that are genetically identical to the PARENT. Sexual REPRODUCTION allows the creation of genetically radically different offspring that are still having the same general flavor (SPECIES). At the molecular level what occurs (wild over simplification alert) is that a pair of Chromosomes bump into one another, exchange chunks of genetic information and drift apart. This is the RECOMBINATION operation, which GA errors generally refer to as CROSSOVER because of the way that genetic material crosses over from one chromosome to another,

The CROSSOVER operation happens in an ENVIRONMENT where the ELECTION of who gets to mate is a function of the FITNESS of the INDIVIDUAL, i.e. how good the individual is at competing in its environment-Some GENETIC ALGORITHMS use a simple function of the fitness measure to select individuals (probabilistically) to undergo genetic operations such as crossover or asexual REPRODUCTION (the propagation of genetic material unaltered). This is fitness-proportionate selection. Other implementations use a model in which certain randomly selected individuals in a subgroup compete and the fittest is selected. This is called tournament selection and is the form of selection we see in nature when stags rut to vie for the privilege of mating with a herd of hinds. The two processes that most contribute to evolution are crossover and fitness based on reproduction.

As it turns out, there are mathematical proofs that indicate that the process of FITNESS proportionate REPRODUCTION is, in fact, near optimal in some senses, MUTATION also plays a role in this process, although how important its role is continues to be a matter of debate (some refer to it as a background operator, while others view it as playing the dominant role in the evolutionary process). It cannot be stressed too strongly that the GENETIC ALGORITHM (as a SIMULATION of a genetic process) is not a random search for a solution to a problem (highly fit INDIVIDUAL), The genetic algorithm uses stochastic processes, but the result is distinctly non-random (better than random) GENETIC ALGORITHMS are used for a number of different application areas.

An example of this would be multidimensional OPTIMIZATION problems in which the character string of the CHROMOSOME can be used to encode the values for the different parameters being optimized. In practice, therefore, we can implement this genetic model of computation by having arrays of bits or characters to represent the CHROMOSOME, Simple bit manipulation operations allow the implementation of CROSSOVER, MUTATION and other operations. Although a substantial amount of research has been performed on variable-length strings and other structures, the majority of work with GENETIC ALGORITHM is focused on fixed-length character strings. We should focus on both this aspect of fixed-lengthiness and the need to encode the representation of the solution being sought as a character string, since these are crucial aspects that distinguish GENETIC PROGRAMMING, which does not have a fixed length representation and there is typically no encoding of the problem,

When the GENETIC ALGORITHM is implemented it is usually done in a manner that involves the following cycle: Evaluate the FITNESS of all of the INDIVIDUALS in the POPULATION. Create a new population by performing operations such as CROSSOVER, fitness-proportionate REPRODUCTION and MUTATION on the individuals whose fitness has just been measured. Discard the old population and iterate using the new population.

One iteration of this loop is referred to as a GENERATION. There is no theoretical reason for this as an implementation model. Indeed, we do not see this punctuated behavior in POPULATIONS in nature as a whole, but it is a convenient implementation model.
The first GENERATION (generation 0) of this process operates on a POPULATION of randomly generated INDIVIDUALS. From there on, the genetic operations, in concert with the FITNESS measure, operate to improve the population.

CHAPTER ONE

STATE OF ART UNDERSTANDING PROBLEM OF GENETIC ALGORITHM FOR SOLVING OPTIMIZATION PROBLEMS

1. GENETIC ALGORITHM

Genetic algorithms use a vocabulary borrowed from natural genetics, a candidate solution is called an individual. Quite often this individual called also truing or chromosome. This might be a little bit misleading; each cell of every organism of a given species carries a certain number of chromosomes, however, we talk about one-chromosome individuals only. Chromosomes are made of units genes arranged in linear *succession; every gene controls the inheritance of one or several characters.*

Each gene can assume a finite number of values, called alleys (feature values). In binary representation chromosome is a vector, consisting of the bits succession, i.e. the succession of zeroes and ones. A set of chromosomes makes a population. A number of chromosomes in population define a population size. The genetic algorithm evaluates a population and generates a new one iteratively, with each successive population referred to as a generation. The population undergoes a simulated evolution, at each generation the relatively "good" solutions reproduce while the relatively "bad" solutions die. To distinguish between different solutions we use an objective (evaluation) function, which plays the role of an environment. Quite often the objective function is called also fitness function.

Due to universe searching ability of genetic algorithm, it is used to solve global external through this chapter. For this reason Genetic Algorithm is widely used to solve different optimization, constructing, predicting and searching problems.

In this chapter I have included the following researches.

1.1. Technological Processes Control By Using Soft Computing

The development of deterministic and fuzzy controllers for technological processes control by using soft computing elements such as Neural, genetic and fuzzy technologies are considered. The learning problems of neural control systems are discussed. The learning algorithm of the recurrent neural network is described. Using learning algorithm and desired time response characteristics of the system the synthesis of neural controller for technological process control is carried out. The results of the simulation of neural control system are described.

Using fuzzy models of control objects and desired time response characteristic of system the synthesis of fuzzy neural controller is carried out. The learning of fuzzy neural controller is performed by using- level procedure and internal arithmetic. The simulation of fuzzy control system is performed and results of simulation are described.

Also the development of the PID controller by using genetic algorithm (GA) is considered. The synthesis of PID controller by the traditional methods requires to posses a great deal of control system knowledge; tuning experience, full information about control object. The use of genetic algorithm (GA) allows automating the tuning process, and does not require having much domain knowledge. Using genetic operators – selection, crossover and mutation operators the tuning of PID controller's coefficients is carried out. The synthesis procedure and result of simulation of control system with PID controller are described.

1.2. Genetic Algorithms As A Searching Algorithms

The theory of natural selection offers some compelling arguments that individuals with certain characteristics are better able to survive and pass on those characteristics to their offspring. A genetic algorithm is a general search procedure based on the ideas of genetics and natural selection, and its power lies in the fact that as members of the population mate, they produce offspring that have a significant chance of retaining the desirable characteristics of their parents, perhaps even combining the best characteristics of both parents. In this manner, the overall fitness of the population can potentially increase from generation to generation as we discover better solutions to our problem.

When applied to the optimization of ANNs for forecasting and classification problems, GAs can be used to search for the right combination of input data, the most suitable forecast horizon, the optimal or near optimal network interconnection patterns and weights among the neurons, and the control parameters (learning rate, momentum rate, tolerance level, etc.), based on the training data used and the pre-set criteria. Like ANNs, GAs do not always guarantee you a perfect solution, but in many cases, it can arrive at an acceptable solution without the time and expense of an exhaustive search.

1.3. Application of Neurofuzzy

NeuroFuzzy systems offer the precision and learning capability of neural networks, and yet are easy to understand like fuzzy systems. Explicit knowledge acquired from experts can be easily incorporated into such a system, and implicit knowledge can be learned from training samples to enhance the accuracy of the output. Furthermore, the modified and new rules can be extracted from a properly trained NeuroFuzzy system, to explain how the results are derived. There are also many other ways to combine neural and fuzzy techniques, to improve the learning speed, adjust learning and momentum rates, etc. Also, newer technologies such as genetic algorithm can be integrated to further enhance the performance of the hybrid systems.

1.4. Binary Search Tree

Searching is a very common operation in most of the management information systems. This can be done efficiently using the binary search technique, which requires an optimal Binary Search Tree (BST) to store the information. The optimality of a BST is determined by the average time taken to search for the information stored in the tree. The failure in search is also to be taken into consideration for optimization. The best known existing solution for optimizing a BST uses the Dynamic Programming technique (DP). DP tries almost all possible binary search trees before finding the optimal BST. This consumes such a vast amount of time even for a moderate sized problem. This urges the necessity for the development of a heuristic algorithm, which would take less time when, compared to DP technique. Genetic Algorithms (GA) are well suited for this task. A Genetic Algorithm use randomness as a tool to guide the search for global optima. This helps it converge in very few iterations when compared to traditional optimization techniques. This paper discusses an implementation of a

genetic algorithm for the construction of an optimal BST. The results obtained are compared with the optimum results found by DP. It is found that the GA outperforms DP for moderate and large sized problems.

1.6. GA Overview

The genetic algorithm performs a parallel, non-comprehensive search for the global maximum of the graph. The search is not precise meaning that there is no guarantee that the global maximum will be found. However, the result should be a good approximation of the maximum value.

Each agent in the population has genes that are represented by bits. In my example, the genes are used to represent a binary number which is translated to an X position on the graph by multiplying the number with an increment value and then adding a base value to the result. In this example the domain is -1.0 <= X <= 2.0. The increment value is determined by calculating the range of X values (ending value minus starting value) and then dividing this by the maximum value of the genes. For example, assuming that the number of genes used by the agents is 22, then the maximum value of the genes is 2^22 - 1. The range of X values, 3.0, is then divided by this number, 4194303, to get the increment, 0.000000715. The base value added to get the X position is simply the starting X value, -1.0.

Four operations are applied to every generation of agents - evaluation, selection, crossover, and mutation. These operations are modeled after the evolutionary process of organisms in nature.

Initially, when the genetic algorithm program is started for the first time or reset, all the agents in the population are given random values for their genes which results in random X values being represented since the genes determine the X position.

CHAPTER TWO

BASIC OF GENETIC ALGORITHMS

2. Basics of Genetic Algorithms

2.1. Basics Of Genetic Algorithms

The three most important aspects of using genetic algorithms are:

(1) Definition of the objective function.

(2) Definition and implementation of the genetic representation.

(3) Definition and implementation of the genetic operators.

Once these three have been defined.

The generic genetic algorithm should work fairly well. Beyond that you can try many different variations to improve performance, find multiple optima (species -if they exist, or parallels the algorithms.

Algorithm GA is

// Start with an initial time

T: =0;

// Initialize a usually random population of individuals

Initpopulation P (t);

// Evaluate fitness of all initial individuals of population

Evaluate P (t);

// Test for termination criterion (time, fitness, etc.)

While not done do

// Increase the time counter

T: = t+ 1,

// Select a sub-population for offspring production

P': = select parents P (t);

// Recombine the "genes" of selected parents

Recombine P' (t);

// Perturb the mated population stochastically

Mutate P' (t),

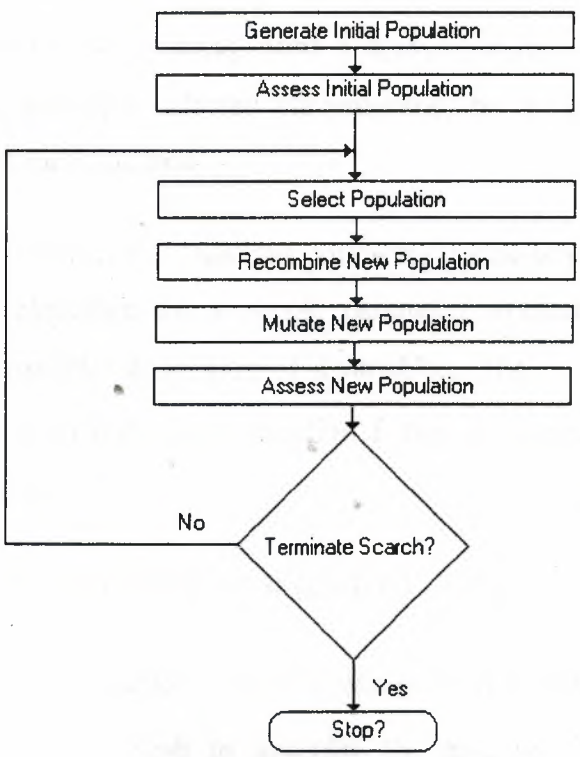
// Evaluate it's new fitness evaluate P' (t);

// Select the survivors from actual fitness

P: = survive P,P'(t);

od

End GA.



2.2. The Genetic Operators

The initial population is chosen at random. GA simulates genetic evolution of a population of tentative solutions (individuals) by means of selection and survival of the fittest, crossover and mutations. Every individual is typically represented as a bit sequence, which makes up its "genetic code". The function to be optimized provides "fitness" values. The structure of a simple genetic algorithm is the same as the structure of any convolution program. During iteration t, a genetic algorithm maintains a population of potential solutions (chromosomes, vectors), G (t)= $\{x_1^t,.....,x_n^t\}$, each solution x_i^t is evaluated to give some measure of its"‘ fitness'" Then, a new population (iteration t+1) is formed by selecting the more fit individuals. Some members of this new population undergo reproduction by means of crossover and mutation, to form new solutions. Crossover combines the features of two parent chromosomes to form two similar offspring by swapping corresponding segments of the parents- For example, if the parents are represented by five-dimensional vectors (a_1,b_1,c_1,d_1,e_1)and (a_2,b_2,c_2,d_2,e_2), then crossing the chromosomes after the second gene would produce the offspring (a_1,b_1,c_2,d_2,e_2)and (a_2,b_2,c_1,d_1,e_1). Mutation arbitrarily after one or more genes of a selected chromosome, by a random change with a probability equal to the mutation rate.

For concrete problem GA has the following block-schema .We discuss the actions of a genetic algorithm for a simple parameter optimization problem. Now suppose we wish to maximize a function of k variables, $f(x_1,.....x_k)$: R^—> R. If the optimization problem is to minimize a function f, this is equivalent to maximizing a function g, where g=-f, i.e.

$$\min\{f(x)\} = \max\{g(x)\} = \{-f\{x\}\}$$

Suppose further that each variable x can take values from a domain $D_i = [a_i, b_i] \subseteq \Re$ and $f(x) \succ 0$ for all x_i. We wish to optimize the function f with some required precision; suppose sex decimal places for the variables' values are desirable,

It is clear that to achieve such precision each domain D_i, should be cut into $(b_i - a_i)*10^6$ equal size ranges. Let us denote by mi the smallest integer such us $(b_i - a_i)*10^6 \le 2m_i - 1$. Then a representation having each variable x_i coded as a binary string of length m_i clearly satisfies the precision requirement. Additionally, the following formula interprets each such string: $x_i = a_i + decimal(1001/0012)$.Where decimal $(string_2)$ represents the decimal value of that binary string. Now, each chromosome (as a potential solution) is represented by a binary string of length $m = \sum_{i=1}^{k} m_i$, the first ml bits map into a value from the range $[a_1, b_1]$, the next group of m_2 bits map into a value from the range $[a_3, b_3]$, and so on, the last group of m_k bits map into a value from the range $[a_k, b_k]$.To initialize a population, we can simply set some p_s number of chromosomes randomly in a bit wise fashion. However, if we have some knowledge about the distribution of potential optima, we may use such information in arranging the set of initial (potential) solutions. The rest of the algorithm is straightforward, in each generation we evaluate each chromosome (using the function f on the decoded sequences of variables), select new population with respect to the probability distribution based on fitness values, and recombine the chromosomes in the new population by mutation and crossover operators. After some number of generations, when no further improvement is observed, the best chromosome represents an (possibly the global) optimal solution. Often we stop the algorithm after a fixed number of iterations depending on speed and resource criteria. For the selection process (selection of a new population with respect to the probability distribution based on fitness values), we must implement the following actions at first, Calculate the fitness value $eval(v_i)$ for each chromosome $v_i (i = 1,....,p_s)$.

• Find the total fitness of the population

$$F = \sum_{i=1}^{p_s} eval(v_i)$$

• Calculate the probability of a selection p_n^2 for each chromosome $v_i (i = 1,....,p_s)$:

$$p_n^i = eval(v_i) / F$$

• Calculate a cumulative probability p_{cum}^i for each chromosome $v_i (i = 1,....,p_s)$:

$$p_{cum}^i = \sum_{j=1}^{i} p_n^j$$

The selection process is implemented p_s times; each time we select a single chromosome for a new population in the following way:

• Generate a random (float) number r from the range [0,1],

• If r < p_{cum}^i then select the first chromosome (v_1); otherwise select the $I - t_j$ chromosome $v_i (2 \le i \le p_s)$ such that

$$p_{cum}^{i-1} < r < p_{cum}^i$$

Obviously, some chromosomes would be selected more than once; the best chromosomes get more copies; the average stay even, and the worst die off. Now we are ready to apply the first recombination operator, crossover, to the individuals in the new population. One of the parameters of a genetic system is probability of crossover p_c. This probability gives us the expected number $p_c p_s$ of chromosomes which undergo the crossover operation. We proceed in the following way:

For each chromosome in the (new) population:

• Generate a random (float) number r from the range [0,1];

• If $r < p_c$, select given chromosome for crossover;

Now we mate selected chromosomes randomly: for each pair of coupled chromosomes we generate random integer number pos from the range [l,m-l] (m is the total length-number of bits - in a chromosome)- The number pos indicate the position of the crossing point. Two chromosomes

$$(b_1 b_2 .. b_{pos} b_{pos+1} .. b_m)$$

$$(c_1 c_2 .. c_{pos} c_{pos+1} .. c_m)$$

are replaced by a pair of their offspring:

$$(b_1 b_2 .. b_{pos} c_{pos+1} .. c_m)$$

$$(c_1 c_2 ... c_{pos} b_{pos+1} .. b_m)$$

The intuition behind the applicability of the crossover operator is information exchange between different potential solutions.

The next recombination operator, mutation, is performed on a bit-by- bit basis.

Another parameter of the genetic system, probability of mutation p_m, gives us the expected number of mutated bits $p_m \cdot m \cdot p_s$.Every bit (in all chromosomes in the whole population) has an equal chance to undergo mutation i.e. change from 0 to1d of vice versa. So we proceed in the following way.

For each chromosome in the current (i.e., after crossover) population and for each bit within the chromosome,

• Generate a random (float) number r from the range [0,1];

• If $r < p_m$ mutate the bit.

The intuition behind the mutation operator is the introduction of some extra variability into the population.

Following selection, crossover, and mutation, the new population is ready for its next evaluation. This evaluation is used to build the probability distribution (for the next selection process). The rest of evolution is just cyclic repetition of the above steps.

However, as it frequently occurs, in earlier generations the fitness values of some chromosomes are better than the value of the best chromosome after a finite number of generations.

It is necessary to note, that classical GA may employ roulette wheel method for selection, which is a stochastic version of the survival of the fittest mechanism. In this method of selection, candidate strings from the current generation G(t) are selected to survive to the next generation G(t=l) by designing a roulette wheel where each string in the population is represented on the wheel in proportion to its fitness value. Thus those strings, which have a high fitness, are given a large share of the wheel, while those strings with low fitness are given a relatively small portion of the roulette wheel. Finally, spinning the roulette wheel p_s times and accepting as candidates those strings, which are indicated at the completion of the spin, make selections,

Example 2.1: As an example, Suppose p_s=5, and consider the following initial population of strings;

G (0)= {(10110),(11000),(11110),(0100I),(00110)}, For each string v_i, in the population, the fitness may be evaluated: $eval(v_i)$. The appropriate share of the roulette wheel to allot the i-th string is obtained by dividing the fitness of the i-th string by the sum of the fatnesses of the entire population:

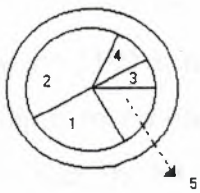
$$eval(v_i) \Big/ \sum_{i=1}^{p_s} eval(v_i)$$

Figure (1.5) shows a listing of the population with associated fitness values and the corresponding roulette wheel.

To compute the next population of strings, the roulette wheel is spun five times [3], The strings chosen by this method of selection, though, are only candidate strings for the next population. Before actually being copied into the new population, these strings must undergo crossover and mutation.

String Fitness Relative (A)

V_i	$eval(v_i)$	Fitness
V_1	10110 2.23	0.14
V_2	11000 7.27	0.47
V_3	11110 1.05	0.07
V_4	01001 3,35	0.21
V_5	00110 1.69	0.11



(B)

In **figure (A)** listing of the five-string population and the associated fitness values, **(b)** Corresponding roulette wheel for string selection.

The integers shown on the roulette wheel correspond to string labels,

110101	1101|01	110100
100100	1001|00	100101
(a)	(b)	(c)

An example (figure) of a crossover for two 6-bit strings,

(a) Two strings are selected for crossover.

(b) A crossover site is selected at random. In this case, k =4,

(c) Now swap the two strings after the k-th bit.

Pairs of the p_s (assume p_s even) candidate strings, which have survived selection, are next chosen for crossover, which is a recombination mechanism. The probability that the crossover operator is applied will be denoted by p_c. Pairs of string are selected randomly from G (t), without replacement, for crossover. A random integer k, called the crossing site, is chosen from {1,2, ... m-1}, and then the tits from the two chosen strings are swapped after the k-th bit with a probability p_c. This process is repeated until G (t) is empty. For example. Figure 11.3. Illustrates a crossover for two 6-bit strings. In this case, the crossing site k is 4, so the bits from the two strings are swapped after the fourth bit.

Finally, after crossover, mutation is applied to the candidate strings. The mutation operator is a stochastic bit-wise complementation applied with uniform probability p_m. That is, for each single bit in the population, the value of the bit is nipped from 0 to 1 or from 1 to 0 with probability p_m. As an example, suppose pm=0. 1, and the string v=11100 is to undergo mutation. The easiest way to determine which bits, if any, to flip is to choose a uniform random number r∈ [0,1] for each bit in the string. If $r \leq p_m$, then the bit is flipped; otherwise, no action is taken. For the string v above, suppose the random numbers (0.91, 0.43, 0.03,0.67,0,29) were generated, and then the resulting mutation is shown bellow. In this case, the third bit was flipped,

Before mutation: 11100

After mutation: 11000

After mutation, the candidate strings are copied into the new population of strings G (t+1), and the whole process is repeated |4|.

2.3. Genetic Algorithm

2.3.1. Basic Description

Genetic algorithms are inspired by Darwin's theory about evolution. Solution to a problem solved by genetic algorithms is evolved.

Algorithm is started with a set of solutions (represented by chromosomes) called population. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected according to their fitness ,the more suitable they are the more chances they have to reproduce.

This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied.

2.3.2. Outline of the Basic Genetic Algorithm

1. [Start] Generate random population of *n* chromosomes (suitable solutions for the problem)
2. [Fitness] Evaluate the fitness *f(x)* of each chromosome *x* in the population
3. [New population] Create a new population by repeating following steps until the new population is complete

    a. [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
    b. [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
    c. [Mutation] With a mutation probability mutate new offspring at each locus (position in chromosome).
    d. [Accepting] Place new offspring in a new population

4. [Replace] Use new generated population for a further run of algorithm
5. [Test] If the end condition is satisfied, stop, and return the best solution in current population
6. [Loop] Go to step 2

Some Comments:

As we can see, the outline of Basic GA is very general. There are many things that can be implemented differently in various problems.

First question is how to create chromosomes, what type of encoding choose. With this is connected crossover and mutation; the two basic operators of GA. Encoding, crossover and mutation are introduced in next chapter.

Next questions are how to select parents for crossover. This can be done in many ways, but the main idea is to select the better parents (in hope that the better parents will produce better offspring). Also you may think, that making new population only by new offspring can cause lost of the best chromosome from the last population. This is true, so so called elitism is often used. This means, that at least one best solution is copied without changes to a new population, so the best solution found can survive to end of run.

Maybe you are wandering, why genetic algorithms do work. It can be partially explained by Schema Theorem (Holland), however, this theorem has been criticized in recent time. If you want to know more, check other resources.

2.4. Operators of GA

As you can see from the genetic algorithm, the crossover and mutation are the most important part of the genetic algorithm. The performance is influenced mainly by these two operators. Before we can explain more about crossover and mutation, some information about chromosomes will be given.

2.4.1. Encoding of a Chromosome

The chromosome should in some way contain information about solution that it represents. The most used way of encoding is a binary string. The chromosome then could look like this:

Chromosome 1	1101100100110110
Chromosome 2	1101111000011110

Each chromosome has one binary string. Each bit in this string can represent some characteristic of the solution. Or the whole string can represent a number - this has been used in the basic GA.

Of course, there are many other ways of encoding. This depends mainly on the solved problem. For example, one can encode directly integer or real numbers, sometimes it is useful to encode some permutations and so on.

2.4.2. Crossover

After we have decided what encoding we will use, we can make a step to crossover. Crossover selects genes from parent chromosomes and creates a new offspring. The simplest way how to do this is to choose randomly some crossover point and everything before this point copy from a first parent and then everything after a crossover point copy from the second parent.

Crossover can then look like this (| is the crossover point):

Chromosome 1	11011	00100110110
Chromosome 2	11011	11000011110
Offspring 1	11011	11000011110
Offspring 2	11011	00100110110

There are other ways to make crossover, for example we can choose more crossover points. Crossover can be rather complicated and very depends on encoding of the encoding of chromosome. Specific crossover made for a specific problem can improve performance of the genetic algorithm.

CHAPTER THREE

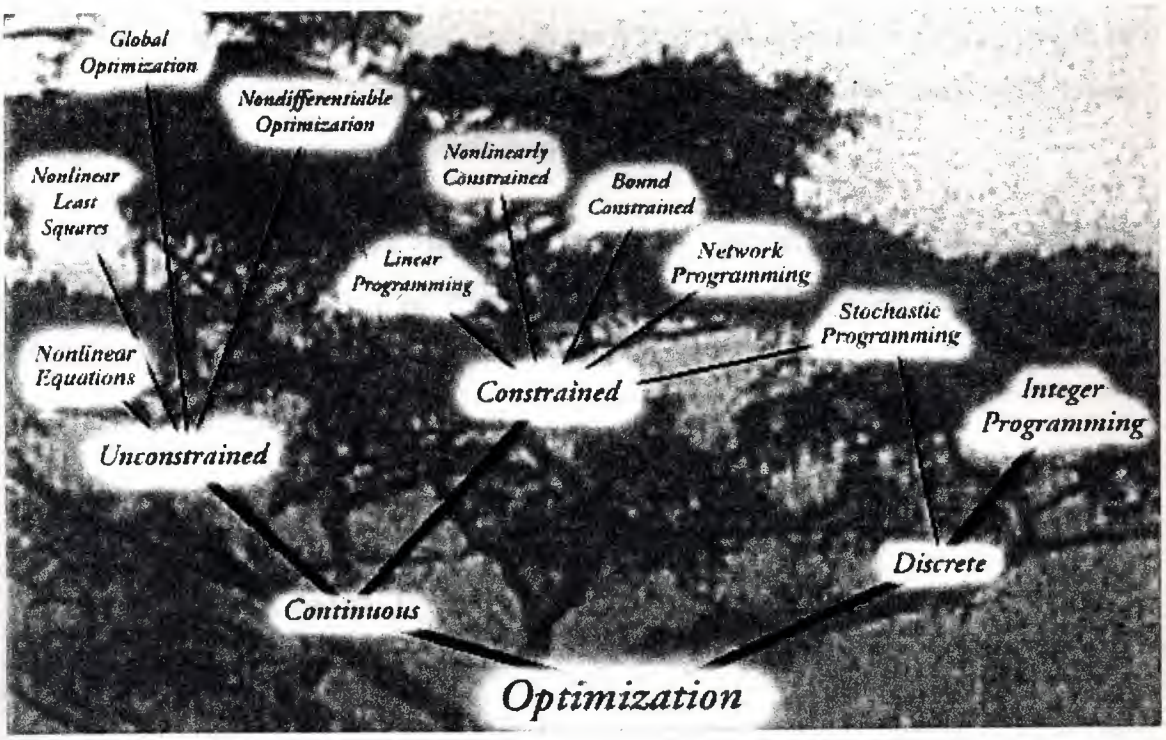
OPTIMIZATION PROBLEM

3. What Is Optimization

Optimization problems are made up of three basic ingredients:

- An **objective function** which we want to minimize or maximize. For instance, in a manufacturing process, we might want to *maximize the profit* or *minimize the cost.* In fitting experimental data to a user-defined model, we might *minimize the total deviation* of observed data from predictions based on the model. In designing an automobile panel, we might want to *maximize the strength.*
- A set of **unknowns** or **variables** which affect the value of the objective function. In the manufacturing problem, the variables might include the *amounts of different resources used* or the *time spent on each activity.* In fitting-the-data problem, the unknowns are the *parameters* that define the model. In the panel design problem, the variables used define the *shape and dimensions* of the panel.
- A set of **constraints** that allow the unknowns to take on certain values but exclude others. For the manufacturing problem, it does not make sense to spend a negative amount of time on any activity, so we constrain all the "time" variables to be non-negative. In the panel design problem, we would probably want to limit the *weight* of the product and to constrain its *shape*.

The Optimization Tree is an online guide to the field of numerical optimization. It introduces the different subfields of optimization and includes outlines of the major algorithms in each area, with pointers to software packages where appropriate. The connections between the Tree's web pages mirrors the relationships between these different areas. Follow the pathways through the tree to see how everything hangs together!



3.1. Multi-Objective Optimization

3.1.1. Introduction

Most realistic optimization problems, particularly those in design, require the simultaneous optimization of more than one objective function. Some examples:

- In bridge construction, a good design is characterized by low total mass and high stiffness.
- Aircraft design requires simultaneous optimization of fuel efficiency, payload, and weight.
- In chemical plant design, or in design of a groundwater remediation facility, objectives to be considered include total investment and net operating costs.
- A good sunroof design in a car could aim to minimize the noise the driver hears and maximize the ventilation.
- The traditional portfolio optimization problem attempts to simultaneously minimize the risk and maximize the fiscal return.

In these and most other cases, it is unlikely that the different objectives would be optimized by the same alternative parameter choices. Hence, some trade-off between the criteria is needed to ensure a satisfactory design.

Multicriteria optimization has its roots in late-nineteenth-century welfare economics, in the works of Edgeworth and Pareto. A mathematical description is as follows:

$$\min_{x \in C} F(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix} \qquad \ldots (MOP)$$

where n >= 2 and

$$C = \{x : h(x) = 0,\ g(x) \leq 0,\ a \leq x \leq b\}$$

denotes the feasible set constrained by equality and inequality constraints and explicit variable bounds. The space in which the objective vector belongs is called the *objective space* and image of the feasible set under **F** is called the *attained set*.

The scalar concept of ``optimality" does not apply directly in the multiobjective setting. A useful replacement is the notion of *Pareto optimality*. Essentially, a vector $x^* \in C$ is said to be Pareto optimal for (MOP) if all other vectors $x \in C$ have a higher value for at least one of the objective functions $f_i(\cdot)$, or else have the same value for all objectives. Formally speaking, we have the following definition:

A point $x^* \in C$ is said to be (glob ally) *Pareto optimal* or a (globally) efficient solution or a non-dominated or a non-inferior point for (MOP) if and only if there is no $x \in C$ such that $f_i(x) \leq f_i(x^*)$ for all $i \in \{1, 2, \ldots, n\}$, with at least one strict inequality.

Pareto optimal points are also known as *efficient*, *non-dominated*, or *non-inferior* points.

We can also speak of *locally* Pareto optimal points, for which the definition is the same as the one just given, except that we restrict attention to a feasible neighborhood of x^*. That is, if $B(x^*, \delta)$ denotes a ball of radius δ around the point

We can also speak of *locally* Pareto optimal points, for which the definition is the same as the one just given, except that we restrict attention to a feasible neighborhood of x^*. That is, if $B(x^*, \delta)$ denotes a ball of radius δ around the point x^*,
we require that for some $\delta > 0$, there is *no* $x \in C \cap B(x^*, \delta)$ such that

$$f_i(x) \leq f_i(x^*), \qquad \text{for all } i = \{1, 2, \ldots, n\}$$

with at least one strict inequality.

Typically, there is an entire curve or surface of Pareto points, whose shape indicates the nature of the tradeoff between different objectives.

3.2. Stochastic Programming

3.2.1. Introduction

All of the model formulations that you have encountered thus far in the Optimization Tree have assumed that the data for the given problem are known accurately. However, for many actual problems, the problem data cannot be known accurately for a variety of reasons. The first reason is due to simple measurement error The second and more fundamental reason is that some data represent information about the future (e.g., product demand or price for a future time period) and simply cannot be known with certainty. We will discuss a few ways of taking this uncertainty into account and, specifically, illustrate how stochastic programming can be used to make some optimal decisions.

3.2.2. Recourse

The fundamental idea behind stochastic linear programming is the concept of *recourse.* Recourse is the ability to take corrective action after a random event has taken place. A simple example of *two-stage recourse* is the following:

- Choose some variables, x, to control what happens today.
- Overnight, a random event happens.
- Tomorrow, take some recourse action, y, to correct what may have gotten messed up by the random event.

We can formulate optimization problems to choose x and y in an optimal way. In this example, there are two periods; the data for the first period are known with certainty and some data for the future periods are stochastic, that is, random.

Example

You are in charge of a local gas company. When you buy gas, you typically deliver some to your customers right away and put the rest in storage. When you sell gas, you take it either from storage or from newly-arrived supplies. Hence, your decision variables are 1) how much gas to purchase and deliver, 2) how much gas to purchase and store, and 3) how much gas to take from storage and deliver to customers. Your decision will depend on the price of gas both now and in future time periods, the storage cost, the size of your storage facility, and the demand in each period. You will decide these variables for each time period considered in the problem. This problem can be modelled as a simple linear program with the objective to minimize overall cost. The solution is valid if the problem data are known with certainty, that is, if the future events unfold as planned.

More than likely, the future will not be precisely as you have planned; you don't know for sure what the price or demand will be in future periods though you can make good guesses. For example, if you deliver gas to your customers for heating purposes, the demand for gas and its purchase price will be strongly dependent on the weather. Predicting the weather is rarely an exact science; therefore, not taking this uncertainty into account may invalidate the results from your model. Your ``optimal" decision for one set of data may not be optimal for the actual situation.

3.2.3. Scenarios

Suppose in our example that we are experiencing a normal winter and that the next winter can be one of three scenarios: normal, cold, or very cold. To formulate this problem as a stochastic linear program, we must first characterize the uncertainty in the

model. The most common method is to formulate scenarios and assign a probability to each scenario. Each of these scenarios has different data as shown in the following table:

Scenario	Probability	Gas Cost ($)	Demand (units)
Normal	1/3	5.0	100
Cold	1/3	6.0	150
Very Cold	1/3	7.5	180

Both the demand for gas and its cost increase as the the weather becomes colder. The storage cost is constant, say, 1 unit of gas is $1per year. If we solve the linear program for each scenario separately, we arrive at three purchase/storage strategies:

- Normal - Normal

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	0	0	500
2	100	0	0	500

- Total Cost = $1000
- Normal - Cold

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	0	0	500
2	150	0	0	900

- Total Cost = $1400
- Normal - Very Cold

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	180	180	1580
2	0	0	0	0

- Total Cost = $1580

We do not know which of the three scenarios will actually occur next year, but we would like our current purchasing decision to put is in the best position to minimize our expected cost. Bear in mind that by the time we make our second purchasing decision, we will know which of the three scenarios has actually happened.

3.3. Formulating a Stochastic Linear Program

Stochastic programs seek to minimize the cost of the first-period decision plus the expected cost of the second-period recourse decision.

Min $c^T x + E_w Q(x, w)$

Subject to $A_x = b$

$x \geq 0$

where

$Q(x, w) = Mind(w)Ty$

Subject to $T(w)x + W(w)y(w) = h(w)$

The first linear program minimizes the first-period direct costs, $\boldsymbol{c^T x}$ plus the expected recourse cost, $Q(x, w)$ over all of the possible scenarios while meeting the first-period constraints, $\boldsymbol{Ax = b}$

The recourse cost Q depends both on x, the first-period decision, and on the random event, w. The second LP describes how to choose y(w) (a different decision for each random scenario). It minimizes the cost $d^T y$ subject to some recourse function, $Tx + Wy = h$. This constraint can be thought of as requiring some action to correct the system after the random event occurs. In our example, this constraint would require the purchase of enough gas to supplement the original amount on hand in order to meet the demand.

One important thing to notice in stochastic programs is that the first-period decision, x, is independent of which second-period scenario actually occurs. This is called the *nonanticipativity property.* The future is uncertain and so today's decision cannot take advantage of knowledge of the future.

3.4. Deterministic Equivalent

The formulation above looks a lot messier than the deterministic LP formulation that we discuss elsewhere. However, we can express this problem in a deterministic for by introducing a different second-period y variable for each scenario. This formulation is called the *deterministic equivalent.*

Min $c^T x + \sum_{i=1}^{N} P_i d_i^T y_i$

Subject to $A_x = b$

$T_{ix} + W_{iyi} = h_i, i = 1,.....,N$

$x \geq 0$

$y_i \geq 0$

where N is the number of scenarios and is the probability of the scenario's occurrence.

For our three-scenario problem, we have

Min $c^T x + P_1 d_1^T y_1 + P_2 d_{21}^T y_2 + P_{31} d_3^T y_3$

S.t. $A_x = b$

$$T_1 x + W_1 y_1 = h_1$$
$$T_2 x + W_2 y_2 = h_2$$
$$T_3 x + W_3 y_3 = h_3$$
$$x, y_i \geq 0$$

Notice that the nonanticipativity constraint is met. There is only one first-period decision, x, whereas there are N second-period decisions, one for each scenario. The first-period decision cannot anticipate one scenario over another and must be feasible for each scenario. That is, $Ax = b$ and $T_i x + W_i y_i = h_i$ for $i = 1,...,N$ Because we solve for all the decisions, x and y_i simultaneously, we are choosing x to be (in some sense) optimal over all the scenarios.

Another feature of the deterministic equivalent is worth noting. Because the T and W matrices are repeated for every scenario in the model, the size of the problem increases linearly with the number of scenarios. Since the structure of the matrices remains the same and because the constraint matrix has a special shape, solution algorithms can take advantage of these properties. Taking uncertainty into account leads to more robust solutions but also requires more computational effort to obtain the solution.

3.5. Comparisons with Other Formulations

Because stochastic programs require more data and computation to solve, most people have opted for simpler solution strategies. One method requires the solution of the problem for each scenario. The solutions to these problems are then examined to find where the solutions are similar and where they are different. Based on this information, subjective decisions can be made to decide the best strategy.

3.5.1. Expected-Value Formulation

A more quantifiable approach is to solve the original LP where all the random data have been replaced with their expected values. Hopefully in this approach we will

do all right on average. For our example then, we consider the (expected value) problem data to be,

Year	Gas cost ($)	Demand
1	5.0	100
2	6.167	143.33

Solving this problem gives the following result,

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	143.33	143.33	1360
2	0	0	0	0

Cost = $1360.00

Let's compute what happens in each scenario if we implement the expected value solution:

Scenario	Recourse Action	Recourse Cost	Total Cost
Normal	Store 43.33 excess @ $1 per unit	43.33	1403.33
Cold	Buy 6.67 units @ $6 per unit	40	1400
Very Cold	Buy 36.67 units @ $7.5 per unit	275	1635

$$\frac{1}{3}1403.33+\frac{1}{3}1400+\frac{1}{3}1635=\$1479.44$$

The expected total cost over all scenarios is

3.5.2 - Stochastic Programming Solution

Forming and solving the stochastic linear program gives the following solution:

Year	Purchase to Use	Purchase to Store	Storage	Cost
1Normal	100	100	100	1100

2Normal	0	0	0	0
2 Cold	50	0	0	300
2 Normal	80	0	0	600

$$\text{Cost} = 1100 + \frac{1}{3}300 + \frac{1}{3}600 = \$1400$$

Similarly we can compute the costs for the stochastic programming solution for each scenario:

Scenario	**Recourse Action**	**Recourse Cost**	**Total Cost**
Normal	None	0	1100
Cold	Buy 50 units @ \$6 per unit	300	1400
Very Cold	Buy 80 units @ \$7.5 per unit	600	1700

The expected total cost over all scenarios is $\frac{1}{3}1100 + \frac{1}{3}1400 + \frac{1}{3}1700 = \$1400$

The difference in these average costs (\$79.44) is the value of the stochastic solution over the expected-value solution. Also notice that the cost of the stochastic solution is greater than or equal to the optimal solution for each scenario solved separately $1100 \geq 1000, 1400 \geq 1400$ and $1635 \geq 1580$. By solving each scenario alone, one assumes perfect information about the future to obtain a minimum cost. The stochastic solution is minimizing over a number of scenarios and, as a result, sacrifices the minimum cost for each scenario in order to obtain a robust solution over all the scenarios.

Conclusion

Randomness in problem data poses a serious challenge for solving many linear programming problems. The solutions obtained are optimal for the specific problem but may not be optimal for the situation that actually occurs. Being able to take this randomness into account is critical for many problems where the essence of the problem

is dealing with the randomness in some optimal way. Stochastic programming enables the modeller to create a solution that is optimal over a set of scenarios.

3.5.3. Solution Techniques

The multiobjective problem is almost always solved by combining the multiple objectives into one scalar objective whose solution is a Pareto optimal point for the original MOP. Most algorithms have been developed in the linear framework (i.e. linear objectives and linear constraints), but the techniques described below are also applicable to nonlinear problems.

3.6. Minimizing Weighted Sums of Functions

A standard technique for MOP is to minimize a positively weighted convex sum of the objectives, that is,

$$\sum_{i=1}^{n} \alpha_i f_i(x), \qquad \alpha_i > 0, \quad i = 1, 2, \ldots, n.$$

It is easy to prove that the minimizer of this combined function is Pareto optimal. It is up to the user to choose appropriate weights. Until recently, considerations of computational expense forced users to restrict themselves to performing only one such minimization. Newer, more ambitious approaches aim to minimize convex sums of the objectives for various settings of the convex weights, therefore generating various points in the Pareto set. Though computationally more expensive, this approach gives an idea of the shape of the Pareto surface and provides the user with more information about the trade-off among the various objectives. However, this method suffers from two drawbacks. First, the relationship between the vector of weights and the Pareto curve is such that a uniform spread of weight parameters rarely produces a uniform spread of points on the Pareto set. Often, all the points found are clustered in certain parts of the Pareto set with no point in the interesting ``middle part" of the set, thereby providing little insight into the shape of the trade-off curve. The second drawback is that non-convex parts of the Pareto set cannot be obtained by minimizing convex combinations of the objectives (note though that non-convex Pareto sets are seldom found in actual applications).

3.7. Homotopy Techniques

Homotopy techniques aim to trace the complete Pareto curve in the bi-objective case (n=2). By tracing the full curve, they overcome the sampling deficiencies of the weighted-sum approach. The main drawback is that this approach does not generalize to the case of more than two objectives. For more information, see Rao and Papalambros and Rakowska, Haftka, and Watson .

3.8. Goal Programming

In the goal programming approach, we minimize one objective while constraining the remaining objectives to be less than given target values. This method is especially useful if the user can afford to solve just one optimization problem. However, it is not always easy to choose appropriate ``goals" for the constraints. Goal programming cannot be used to generate the Pareto set effectively, particularly if the number of objectives is greater than two.

3.9. Normal-Boundary Intersection (NBI)

The normal-boundary intersection method uses a geometrically intuitive parametrization to produce an even spread of points on the Pareto surface, giving an accurate picture of the whole surface. Even for poorly scaled problems (for which the relative scalings on the objectives are vastly different), the spread of Pareto points remains uniform. Given any point generated by NBI, it is usually possible to find a set of weights such that this point minimizes a weighted sum of objectives, as described above. Similarly, it is usually possible to define a goal programming problem for which the NBI point is a solution. NBI can also handle problems where the Pareto surface is discontinuous or non-smooth, unlike homotopy techniques. Unfortunately, a point generated by NBI may not be a Pareto point if the boundary of the attained set in the objective space containing the Pareto points is nonconvex or `folded' (which happens rarely in problems arising from actual applications).

NBI requires the individual minimizers of the individual functions at the outset, which can also be viewed as a drawback.

3.10. Multilevel Programming

Multilevel programming is a one-shot optimization technique and is intended to find just one ``optimal" point as opposed to the entire Pareto surface. The first step in multilevel programming involves ordering the objectives in terms of importance. Next, we find the set of points $x \in C$ for which the minimum value of the first objective function is attained. We then find the points in this set that minimize the second most important objective. The method proceeds recursively until all objectives have been optimized on successively smaller sets.

Multilevel programming is a useful approach if the hierarchical order among the objectives is of prime importance and the user is not interested in the continuous trade-off among the functions. However, problems lower down in the hierarchy become very tightly constrained and often become numerically infeasible, so that the less important objectives have no influence on the final result. Hence, multilevel programming should surely be avoided by users who desire a sensible compromise solution among the various objectives.

3.11. Objective Function

Almost all optimization problems have a single objective function. (When they don't they can often be reformulated so that they do!) The two interesting exceptions are:

- *No objective function.* In some cases (for example, design of integrated circuit layouts), the goal is to find a set of variables that satisfies the constraints of the model. The user does not particularly want to optimize anything so there is no reason to define an objective function. This type of problems is usually called a *feasibility problem.*
- *Multiple objective functions.* Often, the user would actually like to optimize a number of different objectives at once. For instance, in the panel design problem, it would be nice to *minimize weight* and *maximize strength* simultaneously. Usually, the different objectives are not compatible; the variables that optimize one objective may be far from optimal for the others. In practice, problems with multiple objectives are reformulated as single-objective problems by either forming a weighted combination of the different objectives

or else replacing some of the objectives by constraints. These approaches and others are described in our section on multi-objective optimization.

3.12. Unconstrained Optimization

The unconstrained optimization problem is central to the development of optimization software. Constrained optimization algorithms are often extensions of unconstrained algorithms, while nonlinear least squares and nonlinear equation algorithms tend to be specializations. In the *unconstrained optimization problem*, we seek a local minimizer of a real-valued function, f(x), where x is a vector of real variables. In other words, we seek a vector, x*, such that f(x*) <= f(x) for all x close to x*.

Global optimization algorithms try to find an x* that minimizes f over *all possible* vectors x. This is a much harder problem to solve. We do not discuss it here because, at present, no efficient algorithm is known for performing this task. For many applications, local minima are good enough, particularly when the user can draw on his/her own experience and provide a good starting point for the algorithm.

Newton's method gives rise to a wide and important class of algorithms that require computation of the *gradient vector*

$$\nabla f(x) = \begin{pmatrix} \partial_1 f(x) \\ \cdot \\ \cdot \\ \partial_n f(x) \end{pmatrix}$$

and the *Hessian matrix*,

$$\nabla^2 f(x) = \left(\partial_j \delta_i f(x)\right)$$

Although the computation or approximation of the Hessian can be a time-consuming operation, there are many problems for which this computation is justified. We describe algorithms in which the user supplies the Hessian explicitly before moving on to a discussion of algorithms that don't require the Hessian.

Newton's method forms a quadratic model of the objective function around the current iterate x_k. The model function is defined by

$$q_k(\delta) = f(x_k) + \nabla f(x_k)^T \delta + \frac{1}{2}\delta^T \nabla^2 f(x_k)\delta$$

In the basic Newton method, the next iterate is obtained from the minimizer of q_k . When the Hessian matrix, $\nabla^2 f(x_k)$, is positive definite, the quadratic model has a unique minimizer that can be obtained by solving the symmetric $n \times n$ linear system:

$$\nabla^2 f(x_k)\delta_k = -\nabla f(x_k)$$

The next iterate is then

$$x_{k+1} = x_k + \delta_k$$

Convergence is guaranteed if the starting point is sufficiently close to a local minimizer x* at which the Hessian is positive definite. Moreover, the rate of convergence is quadratic, that is,

$$\left\|x_{k+1} - x^*\right\| \le \beta \left\|x_k - x^*\right\|^2$$

for some positive constant β .

In most circumstances, however, the basic Newton method has to be modified to achieve convergence.

These codes obtain convergence when the starting point is not close to a minimizer by using either a *line-search* or a *trust-region* approach.

The line-search variant modifies the search direction to obtain another a downhill, or *descent* direction for f. It then tries different step lengths along this direction until it finds a step that not only decreases f, but also achieves at least a small fraction of this direction's potential.

The trust-region variant uses the original quadratic model function, but they constrain the new iterate to stay in a local neighborhood of the current iterate. To find the step, then, we have to minimize the quadratic subject to staying in this neighborhood, which is generally ellipsoidal in shape.

Line-search and trust-region techniques are suitable if the number of variables is not too large, because the cost per iteration is of order . Codes for problems with a large number of variables tend to use truncated Newton methods, which usually settle for an approximate minimizer of the quadratic model.

So far, we have assumed that the Hessian matrix is available, but the algorithms are unchanged if the Hessian matrix is replaced by a reasonable approximation. Two kinds of methods use approximate Hessians in place of the real thing:

The first possibility is to use difference approximations to the exact Hessian. We exploit the fact that each column of the Hessian can be approximated by taking the difference between two instances of the gradient vector evaluated at two nearby points. For sparse Hessians, we can often approximate many columns of the Hessian with a single gradient evaluation by choosing the evaluation points judiciously.

Quasi-Newton Methods build up an approximation to the Hessian by keeping track of the gradient differences along each step taken by the algorithm. Various conditions are imposed on the approximate Hessian. For example, its behavior along the step just taken is forced to mimic the behavior of the exact Hessian, and it is usually kept positive definite.

Finally, we mention two other approaches for unconstrained problems that are not so closely related to Newton's method:

Nonlinear conjugate gradient methods are motivated by the success of the linear conjugate gradient method in minimizing quadratic functions with positive definite Hessians. They use search directions that combine the negative gradient direction with another direction, chosen so that the search will take place along a direction not previously explored by the algorithm. At least, this property holds for the quadratic

case, for which the minimizer is found exactly within just n iterations. For nonlinear problems, performance is problematic, but these methods do have the advantage that they require only gradient evaluations and do not use much storage.

The nonlinear Simplex method (not to be confused with the simplex method for linear programming) requires neither gradient nor Hessian evaluations. Instead, it performs a pattern search based only on function values. Because it makes little use of information about f, it typically requires a great many iterations to find a solution that is even in the ballpark. It can be useful when f is non smooth or when derivatives are impossible to find, but it is unfortunately often used when one of the algorithms above would be more appropriate.

3.13. NON-LINEAR OPTIMIZATION

3.13.1. Non-linear optimization

The general constrained optimization problem is to minimize a nonlinear function subject to nonlinear constraints. Two equivalent formulations of this problem are useful for describing algorithms. They are

$$\min\{f(x): c_i(x) \le 0, i \in \mathrm{T}, c_i(x) = 0, i \in \varepsilon\}$$

Where each c_i is a mapping from $\Re^n$ to $\Re$, and T and ε are index sets for inequality and equality constraints, respectively; and

$$\min\{f(x); c(x) = 0, l \le x \le u\}$$

Where c maps $\Re^n$ to $\Re^m$, and the lower- and upper-bound vectors, l and u , may contain some infinite components.

The main techniques that have been proposed for solving constrained optimization problems are reduced-gradient methods, sequential linear and quadratic programming methods, and methods based on augmented Lagrangians and exact penalty functions. Fundamental to the understanding of these algorithms is the Lagrangian function, which for formulation (3.1) is defined as:

$$(x,\lambda)\ L = f(x) + \sum_{i \in \tau u \varepsilon} \lambda_i c_i(x)$$

The Lagrange is used to express first-order and second-order conditions for a local minimizer. We simplify matters by stating just first-order necessary and second-order sufficiency conditions without trying to make the weakest possible assumptions.

The first-order necessary conditions for the existence of a local minimizer x^* of the constrained optimization problem (3.1) require the existence of Lagrange multipliers λ_i^*, such that

$$\nabla_x \mathrm{L}(x^*, \lambda^*) = \nabla f(x^*) + \sum_{i \in A^*} \lambda_i^* \nabla c_i(x^*) = 0$$

Where, $$A^* = \{i \in T : c_i(x^*) = 0\} \cup \in$$

Is the *active set* at x^* , and $\lambda_i^* \geq 0$ if $i \in A^* \cap I$. This result requires a constraint qualification to ensure that the geometry of the feasible set is adequately captured by a linearization of the constraints about $x*$. A standard constraint qualification requires the constraint normal, $\nabla c_i(x^*)$ for $i \in A^*$, to be linearly independent.

The second-order sufficiency condition requires that (x^*, λ^*) satisfies the first-order condition and that the Hessian of the Lagrangian

$$\nabla^2{}_{xx} L(x^*, \lambda^*) = \nabla^2 f(x^*) + \sum_{i \in A^*} \lambda_I^* \nabla^2 c_i(x^*)$$

Satisfies the condition

$$w^T \nabla^2{}_{XX} L(x^*, \lambda^*) \omega \rangle 0$$

For all nonzero ω in the set

$$\{w \in \Re^n : \nabla c_i(x^*)^T \omega = 0, i \varepsilon T^* + \cup \varepsilon_1 \nabla c_i(x^*)^T \omega \leq 0, i \in T_0^*\},$$

Where

$$T_+^* = \{i \in A^* \cap T : \lambda_i^* > 0\}, T_0 = \{i \in A^* \cap T : \lambda_I^* = 0\}$$

The previous condition guarantees that the optimization problem is well behaved near x^*; in particular, if the second-order sufficiency condition holds, then x^* is a strict local minimizer of the constrained problem. An important ingredient in the convergence analysis of a constrained algorithm is its behavior in the vicinity of a point (x^*, λ^*) that satisfies the second-order sufficiency condition.

3.13.2. The sequential quadratic programming algorithm

It is a generalization of Newton's method for unconstrained optimization in that it finds a step away from the current point by minimizing a quadratic model of the problem. A number of packages, including NPSOL, NLPQL, OPSYC, OPTIMA, MATLAB, and SQP, are founded on this approach. In its purest form, the sequential QP algorithm replaces the objective function with the quadratic approximation

$$q_k(d) = \nabla f(x_k)^T d + \frac{1}{2} d^T \nabla^2{}_{xx} L(x_k, \lambda_k)^d$$

and replaces the constraint functions by linear approximations. For the formulation (3.1), the step d_k is calculated by solving the quadratic subprogram

$$\min \{q_k(d) : c_i(x_k) + \nabla c_i(x_k)^T d \leq 0, i \in T$$

$$c_i(x_k) + \nabla(x_k)^T d = 0, i \in \varepsilon \}$$

The local convergence properties of the sequential QP approach are well understood when (x^*, λ^*) satisfies the second-order sufficiency conditions. If the starting point x_0 is sufficiently close to x^*, and the Lagrange multiplier estimates $\{\lambda_k\}$ remain sufficiently close to λ^*, then the sequence generated by setting $x_{k+1} = x_k + d_k$ converges to x^* at a second-order rate. These assurances cannot be made

$$(x,\lambda)\ L = f(x) + \sum_{i \in \tau u \varepsilon} \lambda_i c_i(x)$$

The Lagrange is used to express first-order and second-order conditions for a local minimizer. We simplify matters by stating just first-order necessary and second-order sufficiency conditions without trying to make the weakest possible assumptions.

The first-order necessary conditions for the existence of a local minimizer x^* of the constrained optimization problem (3.1) require the existence of Lagrange multipliers λ_i^*, such that

$$\nabla_x \mathrm{L}(x^*, \lambda^*) = \nabla f(x^*) + \sum_{i \in A^*} \lambda_i^* \nabla c_i(x^*) = 0$$

Where, $$A^* = \{i \in T : c_i(x^*) = 0\} \cup \in$$

Is the *active set* at x^* , and $\lambda_i^* \geq 0$ if $i \in A^* \cap I$. This result requires a constraint qualification to ensure that the geometry of the feasible set is adequately captured by a linearization of the constraints about $x*$. A standard constraint qualification requires the constraint normal, $\nabla c_i(x^*)$ for $i \in A^*$, to be linearly independent.

The second-order sufficiency condition requires that (x^*, λ^*) satisfies the first-order condition and that the Hessian of the Lagrangian

$$\nabla^2{}_{xx} L(x^*, \lambda^*) = \nabla^2 f(x^*) + \sum_{i \in A^*} \lambda_I^* \nabla^2 c_i(x^*)$$

Satisfies the condition

$$w^T \nabla^2{}_{xx} L(x^*, \lambda^*) \omega \rangle 0$$

For all nonzero ω in the set

$$\{w \in \Re^n : \nabla c_i(x^*)^T \omega = 0, i \varepsilon T^* + \cup \varepsilon_1 \nabla c_i(x^*)^T \omega \leq 0, i \in T_0^*\}$$

Where

$$T_+^* = \{i \in A^* \cap T : \lambda_i^* > 0\}, T_0 = \{i \in A^* \cap T : \lambda_I^* = 0\}$$

The previous condition guarantees that the optimization problem is well behaved near x^*; in particular, if the second-order sufficiency condition holds, then x^* is a strict local minimizer of the constrained problem. An important ingredient in the convergence analysis of a constrained algorithm is its behavior in the vicinity of a point (x^*, λ^*) that satisfies the second-order sufficiency condition.

3.13.2. The sequential quadratic programming algorithm

It is a generalization of Newton's method for unconstrained optimization in that it finds a step away from the current point by minimizing a quadratic model of the problem. A number of packages, including NPSOL, NLPQL, OPSYC, OPTIMA, MATLAB, and SQP, are founded on this approach. In its purest form, the sequential QP algorithm replaces the objective function with the quadratic approximation

$$q_k(d) = \nabla f(x_k)^T d + \frac{1}{2} d^T \nabla^2{}_{xx} L(x_k, \lambda_k)^d$$

and replaces the constraint functions by linear approximations. For the formulation (3.1), the step d_k is calculated by solving the quadratic subprogram

$$\min \{q_k(d) : c_i(x_k) + \nabla c_i(x_k)^T d \leq 0, i \in T$$

$$c_i(x_k) + \nabla(x_k)^T d = 0, i \in \varepsilon \}$$

The local convergence properties of the sequential QP approach are well understood when (x^*, λ^*) satisfies the second-order sufficiency conditions. If the starting point x_0 is sufficiently close to x^*, and the Lagrange multiplier estimates $\{\lambda_k\}$ remain sufficiently close to λ^*, then the sequence generated by setting $x_{k+1} = x_k + d_k$ converges to x^* at a second-order rate. These assurances cannot be made

in other cases. Indeed, codes based on this approach must modify the sub-problem when the quadratic q_k is unbounded below on the feasible set or when the feasible region is empty.

The Lagrange multiplier estimates that are needed to set up the second-order term in q_k can be obtained by solving an auxiliary problem or by simply using the optimal multipliers for the quadratic sub-problem at the previous iteration. Although the first approach can lead to more accurate estimates, most codes use the second approach.

The strategy based on makes the decision about which of the inequality constraints appear to be active at the solution internally during the solution of the quadratic program. A somewhat different algorithm is obtained by making this decision prior to formulating the quadratic program. This variant explicitly maintains a working set W_k of apparently active indices and solves the quadratic programming problem

$$\min\{q_k(d): c_i(x_k)+\nabla c_i(x_k)^T d = 0, i \in W_k \}$$

To find the step d_k. The contents of W_k updated at each iteration by examining the Lagrange multipliers for the sub-problem and by examining the values of $c_i(x_k+1)$ at the new iterate x_k+1 for $i \notin W_k$. This approach is usually called the EQP (equality-based QP) variant of sequential QP, to distinguish it from the IQP (inequality-based QP) variant described above.

The sequential QP approach outlined above requires the computation of $\nabla^2_{xx}L(x_k,\lambda_k)$. Most codes replace this matrix with the BFGS approximation B_k, which is updated at each iteration. An obvious update strategy (consistent with the BFGS update for unconstrained optimization) would be to define

$$S_K = x_k+1-x_k, \qquad Y_k = \nabla_x L(x_k+1,\lambda_k)-\nabla_x L(x_k,\lambda_k)$$

and update the matrix B_k by using the BFGS formula

$$B_{k+1} = B_k - \frac{B_k 8_k 8_k^T B_k}{8_k^T B_k 8_k} + \frac{Y_k Y_k^T}{Y_k^T 8_k}$$

However, one of the properties that make Broyden-class methods appealing for unconstrained problems-its maintenance of positive definiteness in B_k is no longer assured, since $\nabla^2_{xx} L(x^*, \lambda^*)$ is usually positive definite only in a subspace. This difficulty may be overcome by modifying Y_k. Whenever $Y_k^T 8_k$ is not sufficiently positive, Y_k is reset to

$$Y_k \leftarrow \theta_k Y_k + (1-\theta_k) B_K 8_K ,$$

Where $\theta_k \in [0,1)$ is the number closest to 1 such that $Y_k^T 8_k \geq \sigma 8_k^T B_k 8_k$ for some $\sigma \in (0,1)$. The SQP and NLPQL codes use an approach of this type.

The convergence properties of the basic sequential QP algorithm can be improved by using a line search. The choice of distance to move along the direction generated by the sub-problem is not as clear as in the unconstrained case, where we simply choose a step length that approximately minimizes f along the search direction. For constrained problems we would like the next iterate not only to decrease f but also to come closer to satisfying the constraints. Often these two aims conflict, so it is necessary to weigh their relative importance and define a merit or penalty function, which we can use as a criterion for determining whether or not one point is better than another. The l_1 merit function

$$P_1(x;v) = f(x) + \sum_{i\in\varepsilon} v_i |c_i(x)| + \sum_{i\in T} v_i \max(c_i(x),0),$$

Where $v_i \rangle 0$ are penalty parameters, is used in the NLPQL, MATLAB, and SQP codes, while the augmented Lagrangian merit function

$$L_A(x,\lambda;v) = f(x) + \sum_{i\in\varepsilon} \lambda_i c_i(x) + \frac{1}{2}\sum_{i\in\varepsilon} v_i c_i^2(x) + \frac{1}{2}\sum_{i\in T} \psi_i(x,\lambda;v),$$

Where

$$\psi_i(x,\lambda;v) = \frac{1}{v_i}\{\max\{0, \lambda_i + v_i c_i(x)\}^2 - \lambda_i^2\},$$

Is used in the NLPQL, NPSOL, and OPTIMA codes. The OPSYC code for equality-constrained problems (for which $T = \Phi$) uses the merit function

$$f(x) + \sum_{i \in \varepsilon} \lambda_i c_{i(x)} + \left(\sum_{i \in \varepsilon} v_i c_i^2(x) \right)^{\frac{1}{2}}$$

Which combines features of P_1 and L_A.

An important property of the l_1 merit function is that if (x^*, λ^*) satisfies the second-order sufficiency condition, then x^* is a local minimizer of P_1, provided the penalty parameters are chosen so that $v_i > |\lambda_i^*|$. Although this is an attractive property, the use of P_1 requires care. The main difficulty is that P_1 is not differentiable at any x with $c_i(x) = 0$. Another difficulty is that although x^* is a local minimizer of P_1, it is still possible for the function to be unbounded below. Thus, minimizing P_1 does not always lead to a solution of the constrained problem.

The merit function L_A has similar properties. If (x^*, λ^*) satisfies the second-order sufficiency condition and $\lambda = \lambda^*$, then x is a local minimizer of P_1, provided the penalty parameters v_i are sufficiently large. If $\lambda \neq \lambda^*$, then we can say only that L_A has a minimizer $x(\lambda)$ near x^* and that $x(\lambda)$ approaches x^* as λ converges to λ^*. Note that in contrast to P_1, the merit function L_A is differentiable. The Hessian matrix of L_A is discontinuous at any x with $\lambda_i + v_i c_i(x) = 0$ for $i \in T$, but, at least in the case $T_0^* = \Phi$, these points tend to occur far from the solution.

The use of these merit functions by NLPQL is typical of other codes. Given an iterate x_k and the search direction d_k, NLPQL sets $x_{k+1} = x_k + \alpha_k d_k$, where the step length α_k approximately minimizes $(x_k + \alpha d_k; v)$. If the merit function L_A is selected, the step length α_k is chosen to approximately minimize $L_A(x_k + \alpha d_k, \lambda_k + \alpha(\lambda_{k+1} - \lambda_k); v)$, where d_k is a solution of the quadratic programming sub-problem and λ_{k+1} is the associated Lagrange multiplier.

Where the mapping h is defined implicitly by the equation

$$c[h(x_N), x_N] = 0$$

(We have assumed that the components of have been arranged so that the basic variables come first.) In practice, $x_B = h(x_N)$

Can be recalculated using Newton's method whenever x_N changes. Each Newton iteration has the form

$$x_B \leftarrow x_B - \partial_B c(x_B, x_N)^{-1} c(x_B, x_N),$$

Where $\partial_B c$ is the Jacobian matrix of c with respect to the basic variables. The original constrained problem is now transformed into the bound-constrained problem.

$$\min\{f(h(x_N), x_N) : l_N \le x_N \le u_N\}.$$

Algorithms for this reduced sub-problem subdivide the no basic variables into two categories. These are the *fixed* variables x_F, which usually include most of the variables that are at either their upper or lower bounds and that are to be held constant on the current iteration, and the super basic variables x_S , which are free to move on this iteration. The standard reduced-gradient algorithm, implemented in CONOPT, searches along the steepest-descent direction in the super basic variables. The generalized reduced-gradient codes GRG2 and LSGRG2 use more sophisticated approaches. They either maintain a dense BFGS approximation of the Hessian of f with respect to x_S or use limited-memory conjugate gradient techniques. MINOS also uses a dense approximation to the super basic Hessian matrix. The main difference between MINOS and the other three codes is that MINOS does not apply the reduced-gradient algorithm directly to problem , but rather uses it to solve a linearly constrained sub-problem to find the next step. The overall technique is known as a projected augmented Lagrangian algorithm.

Operations involving the inverse of $\partial_B c(x_B, x_N)$ are frequently required in reduced-gradient algorithms. These operations are facilitated by an LU factorization of

the matrix. GRG2 performs a dense factorization, while CONOPT, MINOS, and LSGRG2 use sparse factorization techniques, making them more suitable for large-scale problems.

When some of the components of the constraint functions are linear, most algorithms aim to retain feasibility of all iterates with respect to these constraints. The optimization problem becomes easier in the sense that there is no curvature term corresponding to these constraints that must be accounted for and, because of feasibility; these constraints make no contribution to the merit function. Numerous codes, such as NPSOL, MINOS and some routines from the NAG (NAG Fortran or NAG C) library, are able to take advantage of linearity in the constraint set. Other codes, such as those in the IMSL, PORT 3, and PROC NLP libraries, are specifically designed for linearly constrained problems. The IMSL codes are based on a sequential quadratic programming algorithm that combines features of the EQP and IQP variants. At each iteration, this algorithm determines a set N_k of near-active indices defined by:

$$N_K = \{i \in \mathrm{T} : c_i(x_K) \geq -r_i\},$$

Where the tolerances r_i tend to decrease on later iterations. The step d_K is obtained by solving the sub-problem.

$$\min\{q_K(d) : c_i(x_K + d_K) = 0, i \in \varepsilon, c_i(x_K + d_K) \leq c_i(x_K), i \in N_K\},$$

Where

$$q_K(d) = \nabla f(x_K)^T d + \frac{1}{2} d^T B_K d,$$

And B_K is a BFGS approximation to $\nabla^2 f(x_K)$. This algorithm is designed to avoid the short steps that EQP methods sometimes produce, without taking many unnecessary constraints into account, as IQP methods do.

3.13.5. Feasible sequential quadratic programming algorithms

Finally, we mention feasible sequential quadratic programming algorithms, which, as their name suggests, constrain all iterates to be feasible. They are more

expensive than standard sequential QP algorithms, but they are useful when the objective function f is difficult or impossible to calculate outside the feasible set, or when termination of the algorithm at an infeasible point (which may happen with most algorithms) is undesirable. The code FSQP solves problems of the form

$$\min\{f(x) : c(x) \leq 0, Ax = b\}$$

In this algorithm, the step is defined as a combination of the sequential QP direction, a strictly feasible direction (which points into the interior of the feasible set) and, possibly, a second-order correction direction. This mix of directions is adjusted to ensure feasibility while retaining fast local convergence properties. Feasible algorithms have the additional advantage that the objective function f can be used as a merit function, since, by definition, the constraints are always satisfied. FSQP also solves problems in which f is not itself smooth, but is rather the maximum of a finite set of smooth functions

$f_i : \Re^n \to \Re$

CHAPTER FOUR

GENETIC ALGORITHMS BASED ON OPTIMIZATION

4. Optimization based on Genetic Algorithms

Genetic algorithms were formally introduced in the United States in the 1970s by John Holland at University of Michigan. The continuing price/performance improvements of computational systems have made them attractive for some types of optimization. In particular, genetic algorithms work very well on mixed (continuous *and* discrete), combinatorial problems. They are less susceptible to getting 'stuck' at local optima than gradient search methods. But they tend to be computationally expensive.

To use a genetic algorithm, you must represent a solution to your problem as a genome (or chromosome). The genetic algorithm then creates a population of solutions and applies genetic operators such as mutation and crossover to evolve the solutions in order to find the best one.

This presentation outlines some of the basics of genetic algorithms. The three most important aspects of using genetic algorithms are:

(1) Definition of the objective function.

(2) Definition and implementation of the genetic representation.

(3) Definition and implementation of the genetic operators

Once these three have been defined, the generic genetic algorithm should work fairly well. Beyond that you can try many different variations to improve performance, find multiple optima (species - if they exist), or parallelism the algorithms,

Genetic algorithm (GA) uses the principles of evolution, natural selection, and genetics from natural biological systems in a computer algorithm to simulate evolution. Essentially, the genetic algorithm is an optimization technique that performs a parallel, stochastic, but directed search to evolve the most fit population. In this section we will introduce the genetic algorithm and explain how it can be used for design of fuzzy

systems. The genetic algorithm borrows ideas from and attempts to simulate Darwin's theory on natural selection and Mendel's work in genetics on inheritance. The genetic algorithm is an optimization technique that evaluates more than one area of the search space and can discover more than one solution to a problem. In particular, it provides a stochastic optimization method where if it "gets stuck" at a local optimum, it tries to simultaneously find other parts of the search space and jump out" of the local optimum to a global one.

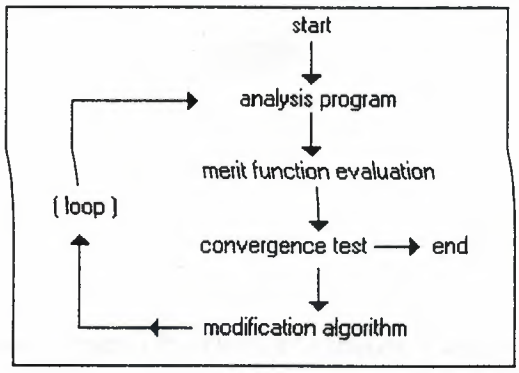


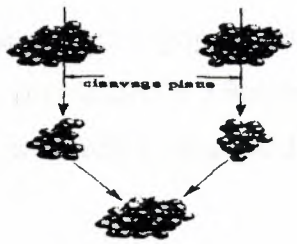
Figure 4.1 Characteristics common to all optimizers

4.1. Genetic Algorithm Structural Optimization

Atomistic models of materials can provide accurate total energies. For problems where the structures are not known, however, discovering the lowest energy geometry is difficult. This is particularly true for atomic clusters, whose structure may vary dramatically with a small change in the number of atoms. For this type of problem, the number of possible stable structures increases exponentially fast with the number of atoms. Furthermore, there is considerable experimental difficulty in determining the structure of an atomic cluster. We have been able to address this problem using a novel approach to applying genetic algorithms. The Darwinian evolution process inspires these algorithms. A population of structures is maintained, and "mating" structures and selecting out the lowest energy geometries produce new generations,

The key to a successful genetic algorithm is to design a mating process that allows for the good parts of the parent structures to be inherited by the next generation. Such a process allows for efficient searching of the possible stable structures. A poor mating algorithm is no better than a random search. We have designed a new mating

process, depicted at left. Two structures are chosen as "parent" structures, Each one is divided into two halves by a cleavage plane. A new structure is generated by connecting half of each parent into a new cluster, followed by atomic relaxation to a local minimum.



4.2. Genetic Algorithm

4.2.1. Basic Description

Genetic algorithms are inspired by Darwin's theory about evolution. Solution to a problem solved by genetic algorithms is evolved.

Algorithm is started with a set of solutions (represented by chromosomes) called population. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected according to their fitness ,the more suitable they are the more chances they have to reproduce.

This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied.

4.2.2. Outline of the Basic Genetic Algorithm

1. [Start] Generate random population of n chromosomes (suitable solutions for the problem)
2. [Fitness] Evaluate the fitness $f(x)$ of each chromosome x in the population
3. [New population] Create a new population by repeating following steps until the new population is complete
4. [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)

5. [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
6. [Mutation] With a mutation probability mutate new offspring at each locus (position in chromosome).
7. [Accepting] Place new offspring in a new population
8. [Replace] Use new generated population for a further run of algorithm
9. [Test] If the end condition is satisfied, stop, and return the best solution in current population
10. [Loop] Go to step 2

Some Comments:

As you can see, the outline of Basic GA is very general. There are many things that can be implemented differently in various problems.

First question is how to create chromosomes, what type of encoding choose. With this is connected crossover and mutation; the two basic operators of GA. Encoding, crossover and mutation are introduced in next chapter.

Next questions are how to select parents for crossover. This can be done in many ways, but the main idea is to select the better parents (in hope that the better parents will produce better offspring). Also you may think, that making new population only by new offspring can cause lost of the best chromosome from the last population. This is true, so so called elitism is often used. This means, that at least one best solution is copied without changes to a new population, so the best solution found can survive to end of run.

Maybe you are wandering, why genetic algorithms do work. It can be partially explained by Schema Theorem (Holland), however, this theorem has been criticized in recent time. If you want to know more, check other resources.

4.3. Operators of GA

As you can see from the genetic algorithm, the crossover and mutation are the most important part of the genetic algorithm. The performance is influenced mainly by these two operators. Before we can explain more about crossover and mutation, some information about chromosomes will be given.

4.3.1. Encoding of a Chromosome

The chromosome should in some way contain information about solution that it represents. The most used way of encoding is a binary string. The chromosome then could look like this:

Chromosome 1	1101100100110110
Chromosome 2	1101111000011110

Each chromosome has one binary string. Each bit in this string can represent some characteristic of the solution. Or the whole string can represent a number - this has been used in the basic GA.

Of course, there are many other ways of encoding. This depends mainly on the solved problem. For example, one can encode directly integer or real numbers, sometimes it is useful to encode some permutations and so on.

4.3.2. Crossover

After we have decided what encoding we will use, we can make a step to crossover. Crossover selects genes from parent chromosomes and creates a new offspring. The simplest way how to do this is to choose randomly some crossover point and everything before this point copy from a first parent and then everything after a crossover point copy from the second parent.

Crossover can then look like this (| is the crossover point):

Chromosome 1	11011 | 00100110110
Chromosome 2	11011 | 11000011110

Offspring 1	11011 | 11000011110
Offspring 2	11011 | 00100110110

There are other ways to make crossover, for example we can choose more crossover points. Crossover can be rather complicated and very depends on encoding of the encoding of chromosome. Specific crossover made for a specific problem can improve performance of the genetic algorithm.

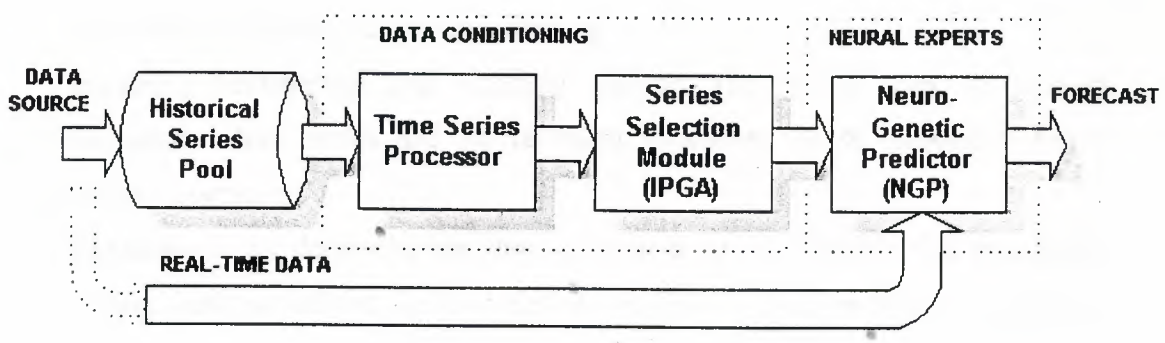
CHAPTER FIVE

GENETIC ALGORITHM FOR FORECASTING PROBLEMS

5. Forecasting Approach

Nodes has implemented an efficient predictive tool to be used for financial applications, combining up-to-date artificial intelligence technologies such as genetic algorithms and neural networks. It integrates a wide variety of data preprocessing, visualization and control facilities in a modular, user-friendly environment.

It is ideally suited for financial time series prediction, allowing automatic input data selection, multiple neural forecasting approaches, and various output data charting styles. Statistical analysis enables the computation of specific performance measures for the predicted values and the estimation of their associated confidence intervals. The system logs on disk all important information related to the current experiment.



The general structure of the system in the figure above includes the following modules:

- historical database:it consists of a pool of economical and technical indicators time series, obtained from a high quality data source like Datastream, VWD.

- Time Series Processor (TSP):it provides input/target series management, basic pre/post-processing facilities, suitable temporal synchronization.
- time series selection module (IPGA): significant input data for the forecasting module is selected using a perform ant artificial intelligence approach based on genetic algorithms and neural networks. The module extracts from the overall pool a particular subset of properly processed series.
- Neuro-Genetic Predictor (NGP): it uses a hybrid neuro-genetic learning strategy to model the complicated relationships that may exist between the input data provided by the selection module and the specific target series. Up-to-date techniques are employed in order to asses the generalization capability of the resulting networks, and the performances are reported in terms of consecrated figures of merit.

The main features of the nodes forecasting system are:

- multiple data source import: Datastream, VWD
- simple and flexible interface with the primary data source: individual time series may be loaded and removed, temporally synchronized by means of custom defined lags, and saved on disk along with temporal information
- data pre- and post-processing facilities: detrendization, normalization, outlier suppression, Fourier and wavelet filtering
- enhanced fundamental and technical analysis: the system uses 68 primary economical time series and 35 technical indicators, to be processed by 34 distinct operations
- significant input data selection through genetic search: flexible GA parameters setting, custom defined fitness function, multiple evaluation schemes, definable constraints
- performant forecasting using multiple neural networks architectures (MLP, Elman, modular and generalized feedforward)
- easy run-time control: suspend/resume button, auto-resume
- off-line statistical analysis of experiments: estimation of the confidence intervals of predicted values, input pruning, "sanity check" of networks used (model validation tests, weights histogram)

- important info saved on disk: genetically selected series, the main performance measures (MSE, directional change, path) and the parameters of the experiment are saved on separate files enabling easy startup configuring and repeatability.

5.1. Genetic Algorithms and Genetically Evolved Neural Networks

5.1.1. An Introduction to Genetic Algorithms

Many studies have shown that ANNs have the capability to learn the underlying mechanics of time series, or, in the case of trading applications, the market dynamics. However, it is often difficult to design good ANNs, because many of the basic principles governing information processing in ANNs are hard to understand, and the complex interactions among network units usually makes engineering techniques like divide and conquer inapplicable. When complex combinations of performance criteria (such as learning speed, compactness, generalization ability, and noise resistance) are given, and as network applications continue to grow in size and complexity, the human engineering approach will not work and a more efficient, automated solution will be needed.

GA is reminiscent of sexual reproduction in which the genes of two parents combine to form those of their children. When it is applied to problem solving, the basic premise is that we can create an initial population of individuals representing possible solutions to a problem we are trying to solve. Each of these individual has certain characteristics that make them more or less fit as members of the population. The most fit members will have a higher probability of mating than the less fit members, to produce offspring that have a significant chance of retaining the desirable characteristics of their parents. This method is very effective at finding optimal or near optimal solutions to a wide variety of problems, because it does not impose many of the limitations required by traditional methods. It is an elegant generate and test strategy that can identify and exploit regularities in the environment, and converges on solutions that were globally optimal or nearly so.

GA have been increasingly applied in ANN design in several ways: topology optimization, genetic training algorithms and control parameter optimization. In topology optimization, GA is used to select a topology (number of hidden layers,

number of hidden nodes, interconnection pattern) for the ANN which in turn is trained using some training scheme, most commonly back propagation. In genetic training algorithms, the learning of a ANN is formulated as a weights optimization problem, usually using the inverse mean squared error as a fitness measure. Many of the control parameters such as learning rate, momentum rate, tolerance level, etc., can also be optimized using GA. In addition, GA have been used in many other innovative ways, for instance, creating new indicators based on existing ones, selecting good indicators, to evolve optimal trading systems and to complement other techniques such as fuzzy logic.

There are four stages in the genetic search process: initialization, evaluation and selection, crossover and mutation. In the initialization stage, a population of genetic structures which are randomly distributed in the solution space is selected as the starting point of the search.

In the second stage, each structure is evaluated using a fitness function and assigned a fitness value. On the basis of their relative fitness values, structures in the current population are selected for reproduction. A stochastic procedure ensures that the expected number of offspring associated with a given structure s is u(s)/u(P), where u(s) is the observed performance of s and u(P) is the average performance of all structures in the current population. Thus structures with high performance are more likely to be chosen for replication while poor performing structures are eventually removed from the population. In the absence of other mechanisms, such a selective process would cause the best performing structures in the initial population to occupy an increasingly larger proportion of the population over time.

The selected structures are recombined using crossover, with two complementary search functions. First, it provides new points for further testing of structures already present in the population; Secondly, it introduces instances of new structures into the population.

Generally, crossover draws only on the information present in the solutions of the current population in generating new solutions for evaluation. If specific information is missing (due to storage limitation or loss incurred during the selection process of a previous generation), then crossover is unable to produce new structures that contain

this piece of information. A mutation operator, which arbitrarily alters one or more components of a selected structure, provides the means for introducing new information into the population. However, mutation functions as a background operator with a very low probability of application. The presence of mutation ensures that the probability of reaching any point in the search space is never zero.

5.2. Genetically Evolved Neural Networks

One method of automating ANN architecture design using GA is described below. It comprises two adaptive processes: genetic search through input data window, forecast horizon, network architecture space and control parameters to select the best performers, and backpropagation learning in individual networks to evaluate the selected architectures.

The method begins with an initial population of randomly generated networks which are represented by overlapped tree structures as illustrated in Fig. 1. and 2 below.

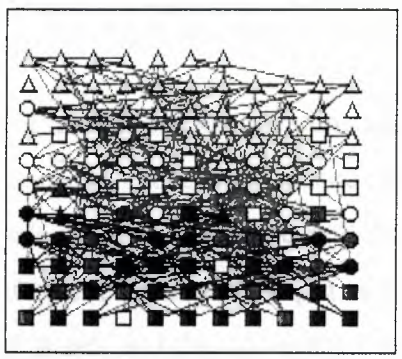


Figure 5.1: An initial population of networks.

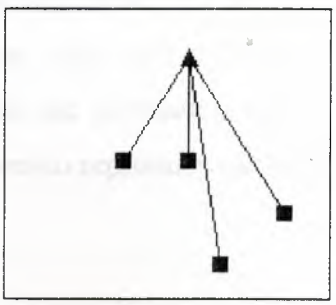


Figure 5.2: One of the randomly generated network.

In Fig. 5.1 and 5.2, each rectangle represents an input node and the triangle represents an output node. The networks will grow and their hidden nodes will be inserted into the networks as the population evolves. The input nodes take a random combination of input data from an input data window which picks up data items from the training data file and supplies them to the network. The output node randomly selects a forecast horizon from an output window and uses the associated data item as target value. Both windows slide down the training data file sequentially or randomly. Fig. 3 illustrates an input window of size 3, and an output window with forecast horizons ranging from 1 to 3 steps ahead.

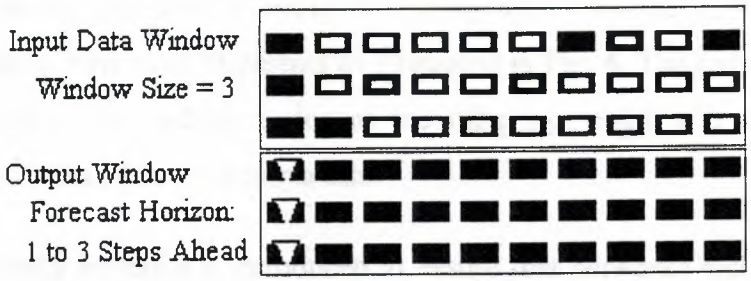


Figure 5.3: The initial setting of the input and output windows used by all the networks.

The initial population of networks then goes through the first evolution cycle. As in real biological systems, learning cycles are nested within cycles of evolution in populations. Each learning cycle involves the entire population of networks with the set of input output pairs provided by the input and output windows. The networks' outputs are compared with the desired target, and the connection weights are adjusted to achieve the desired input output mapping by minimizing the errors.

At the end of each learning cycle, the networks are ranked according to some pre-determined criteria such as their generalization capability. The poor-performing networks will be removed from the population, while the fitter ones are retained and selected for the crossover process to reproduce the offspring for the next generation.

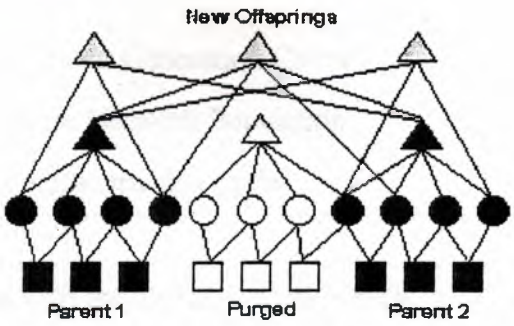


Figure 5.4: Mating of two fittest parent networks to produce new offspring.

There are several ways to cross the parent nets. One way is to produce new offspring by mating two most fit parents as illustrated in Fig. 4. The output nodes of the parents are used as the hidden nodes of the offspring which will then inherit the knowledge already acquired by their parents.

Occasionally mutation is introduced to ensure that networks will not be trapped in local minima during the learning process. One way to mutate is to randomize the weights of those lowly ranked networks, change their input combination in the input data window and/or their forecast horizon.

After each evolution cycle, an image of the fittest network is kept. The image includes the current input data combination, forecast horizon, interconnection patterns and weights of the fittest network, such that the fittest network can go through the next evolution and training cycles together with the rest of the newly formed population.

5.3. Training Cycle

- Increment Iteration Count
- Neural Net Learning
- Check Criteria To Stop? If Yes --> Stop
    - Evaluate Network Population
    - Rank Network Population
    - Store Image of Fittest Network
    - Select Most Fit Parents
    - Crossover & Mutation

- Increment Generation Count
- Update Network Population
- Continue training cycle

Fig. 5. The training cycle with the nested evolution cycle.

The image will remain intact during subsequent evolution cycles until another fittest network emerges. A complete evolution cycle, with the nested training cycle, is illustrated in Fig. 5.

5.4. An Application Example

In this example, the following fundamental data of a stock were used as training data to forecast its future average price per share. The input data window size is 3 and the range of the forecast horizon is 1 to 3 steps.

Year	Avg$/Sh	Sls/Sh	CF/Sh	Ern/Sh	Div/Sh%	Cap$/Sh	BV/Sh	Avg P/E	RelP/E	Div
1979	4.61	9.21	0.69	0.53	0.23	0.48	2.68	4.4	15.8	1.2
1980	6.53	9.27	0.7	0.51	0.25	0.51	2.94	8.7	1.26	5.1
1981	7.44	10.29	0.84	0.61	0.26	0.38	3.29	12.8	1.7	3.9
1982	7.37	9.66	0.92	0.63	0.3	0.39	3.63	12.2	1.48	3.5
1983	11.03	10.32	1.05	0.74	0.35	0.25	4.05	11.7	1.29	4.1
1984	12.46	11.52	1.22	0.89	0.4	0.41	4.59	14.9	1.26	3.2
1985	12.72	11.42	1.13	0.8	0.43	0.39	4.99	14	1.3	3.2
1986	13.78	12.98	1.3	0.83	0.5	0.55	5.27	15.9	1.29	3.4
1987	15.79	14.12	1.36	0.94	0.53	0.7	5.76	16.6	1.13	3.6
1988	14.72	11.82	1.14	0.8	0.6	0.7	4.11	16.8	1.12	3.4
1989	13.75	13.28	1.26	0.87	0.62	0.58	4.1	18.4	1.53	4.5

After 30 generations, the fittest network, as shown in Fig. 6, emerged. The number of input nodes shown in Fig. 5.7 was less than that of Fig. 5.6, because some of those in Fig. 6 referred to the same input nodes in Fig. 5.7.

The result, as shown in Fig. 5.7, indicated that a window size of 2 is sufficient, and that the best forecast can be achieved with a forecast horizon of 1 step ahead. The output of the fittest network is shown in Fig. 5.8. The downward turn detected at the end of the training data indicates a strong sell signal which was confirmed by the actual target values.

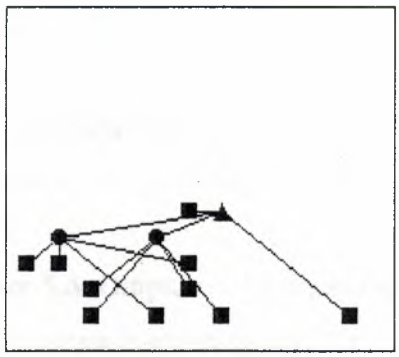


Figure 5.6. The best network after 30 generations of evolution.

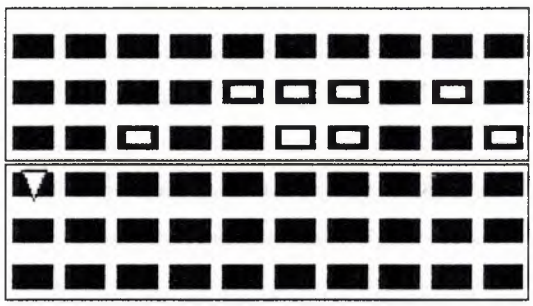


Figure 5.7. The input data combination of the fittest network with forecast horizon of 1 step ahead.

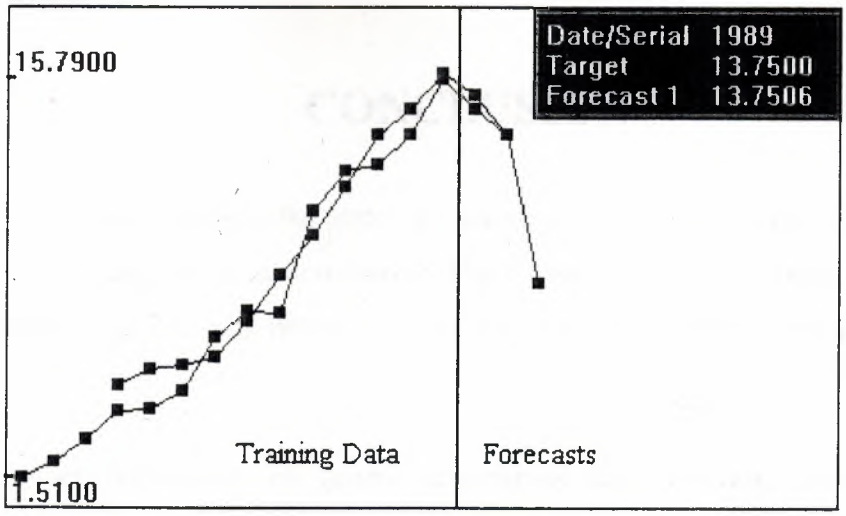


Figure 5.8. Output of the fittest network.

One can also fix the desired forecast horizon to say, 2 or 3 steps ahead, but the forecast will not be as good as that of 1 step in this problem.

CONCLUSION

First of all in this project the state of understanding the art of genetic algorithms for solving genetic algorithms is considered .The Crossover and Mutation are the two basic parameters of genetic algorithms .In Encoding the crossover these parameters are used.

The other parameters of genetic algorithms are Selection, Encoding and Population-size.

The applications of genetic algorithms are used to solve the NP-hard problems for machine learning and also for evolving simple problems. The main features of the genetic algorithms are based on optimization and they are represented by Multimodal functions and Simple trading models.

The problems are also solved by using the Non-linear optimization. Solving the non-linear optimization the algorithms were used that are as follows:

a) The Sequential Quadratic programming algorithm.
b) Augmented Langragian algorithm.
c) Reduced-gradient algorithm.
d) Feasible Sequential Quadratic Programming algorithm.

The main techniques used for solving constrained optimization problems are written above. By using these algorithms we solved the optimization problems.

The Appendices were used in these algorithms that are as follows:

1) Conopt
2) GRG2
3) Lancelot
4) Mat-lab Optimization Toolbox
5) Minos etc.

REFERENCES

[1] Rahib Abiyev, Softcomputing elements based controllers. Electrical, Electronics and Computer Engineering Symposium NEU-CEE2001 & Exhibition, Nicosia, TRNC, Turkey, 2001.

[2] Rahib Abiyev. "Genetic Algorithm Synthesis of the Industrial Controllers". Pakistan Journal of Applied Science. Volume 1, N:3, June, 2001,pp.283-286.

[3] M. Aiken, "Forecasting T-Bill Rates with a Neural Network," Technical Analysis of Stocks and Commodities, Vol. 13, No. 5, May 1995, pp. 85-88.

[4] M. Aiken, Jay Krosp, Chitti Govindarajulu, M. Vanjani, and Randy Sexton, "A Neural Network for Predicting Total Industrial Production," Journal of End User Computing, 7(2), Spring 1995, 19-23.

[5] M. Aiken, "Artificial Neural Systems as a Research Paradigm for the Study of Group Decision Support Systems," Group Decision and Negotiation, in press.

[6] Morris Stocks, T. Singleton, and M. Aiken,"Forecasting Bankruptcies Using a Neural Network," International Business Schools Computing Quarterly, Spring 1995, Vol. 7, No. 1, pp 32-35.

[7] Kelly Fish, James Barnes, M. Aiken,"Artificial Neural Networks: A New Methodology for Industrial Market Segmentation," Industrial Marketing Management, Vol. 24, No. 5, 1995

[6] Tommie Singleton, Morris Stocks, and M. Aiken,"Using a Group Decision Support System for Financial Decision Making," SEDSI Conference, March 1995.

[7] M. Stocks, T. Singleton, M. Aiken,"Bankruptcy Prediction: Logistic Analysis vs. an Artifical Neural Network," 1995 Southwest Decision Sciences Institute Conference, March 1-4, 1995, Houston, TX.

[8] M. Aiken, J. Morrison, J. Paolillo, and L. Motiwalla,"Forecasting Gross Domestic Product Using a Neural Network," Decision Sciences Conference, November 1995.

[9] Wong, F. & Tan C., "Hybrid Neural, Genetic and Fuzzy Systems," in Trading on the Edge (Deboeck G. Ed.), John Wiley & Sons, Inc., 1994.

[10] Peters E., Chaos and Order in the Capital Markets, published by John Wiley & Son, 1992.

[11] Peters E., Fractal Structure in the Capital Markets, Financial Analyst Journal, May/June 93 Issue.

[12] Peters E., Fractal Market Analysis, John Wiley & Sons, Inc., 1994. Holland, J. (1975). Adaptation in natural and artificial systems.

[13] Tan P., "Automatic selection of neural network architectures via genetic algorithm", MSc thesis in preparation.

[14] Tan P., Lim G., Chua K., Wong F, Neo S."A Comparative Study Among Neural Networks, Radial Basis Functions and Regression Models", Second International Conference on Automation, Robotics and Computer Vision (ICARCV'92), Sept., 1992, Singapore.

[15] Wong, F., "NeuroFuzzy Computing Technology," Guest Editorial, NeuroVe$t Journal, May-Jun. 1994, pp8-10.

[16] Wong, F., Wang, P., Goh T., Quek B.K., "A Fuzzy Neural System for Stock Selection," Financial Analyst Journal, published by Association for Investment Management and Research, Jan Feb,. 1992.

[17] Wong F., "Time Series Forecasting Using BackPropagation Neural Networks," Neurocomputing, 2 (1990/91) 147 159, Elsevier.

[18] Wong F., "FastProp: A Selective Training Algorithm For Fast Error Propagation," Proceedings of the IJCNN, 1991, Singapore, pp 2038 2044.

[19] Wong F., Wang P., Goh T., "Fuzzy Neural Systems For Decision Making," Proceedings of the International Joint Conference on Neural Networks (IJCNN'91), Nov 1991, Singapore.

[20] Wong F., Tan P., "Neural Networks and Genetic Algorithm For Economic Forecasting", AI in Economics and Business Administration, Ed. I.H. Daniels & Feelders, Netherland.

[21] Wong F., Tan P., Zhang X., "Neural Networks, Genetic Algorithms and Fuzzy Logic for Forecasting," Proceedings of the Third International Conference on Advanced Trading Technologies AI Applications on Wall Street & Worldwide, New York, Marriott Financial Center, July 1992.

[22] Wong F., "An Integrated Neural Network For Financial Time Series Analysis", Proceedings of the 24th Hawaii International Conference on System Sciences, Jan. 1991.

[23] Wong F., "NeuroForecaster -- A Neural Network For Time Series Predictive Analysis," Technical Report, National University of Singapore, May 1990.

[24] Wong F., Tan P., "Neural Networks and Genetic Algorithm For Economic Forecasting", to appear in AI in Economics and Business Administration,

Ed. I.H. Daniels & Feelders, Netherland.

[25] Wong F., Lee D., "A Hybrid Neural Network For Stock Selection," Proceedings of The Second Annual International Conference on ARTIFICIAL INTELLIGENCE APPLICATIONS ON WALL STREET, April 19-22, 1993, New York, NY USA.

[26] Wong F., "Hybrid Systems of Neural Network, Fuzzy Logic and Genetic Algorithms", in Advanced Technology for Trading, Portfolio and Risk Management, Edited by Dr. Guido Deboeck, Advanced Analytical Laboratory, Investment Dept., World Bank.

[27] Wong F., "Time Series Forecasting Using Backpropagation Neural Network", Neurocomputing, Elsevier,1990, 147-159

[28] Wong F. and Wang P., "A stock selection strategy using fuzzy neural networks", Neurocomputing 2 Elsevier (1990/91) 233-242

[29] Wong F. and Wang P.Z., "A Fuzzy Neural Network Approach For Forex Investment," International Fuzzy Engineering Symposium '91, Nov. 1991, Japan.

[30] Wong, F., "NeuroGenetic Computing ," NeuroVe$t Journal, July-August 1994.

[31] Kamijo K. and Tanagawa T., "Stock price pattern recognition. A recurrent neural network approach", Proc. International Joint Conference on Neural Networks, San Diego, June 1990, I 215 221.

[32] Lapedes, A., R. Farber (1987). "How neural nets work" Advances in Neural Information Systems.

[33] Parker D., "Learning Logic", Report TR 47, MIT Center for Computational Research in Economics and Management Science.

[34] Rumelhart D., Hinton G. & Williams R., Parallel Distributed Processing, 1986. Cambridge, MA, MIT Press.

[35] Trippi R. & Lee J., State-of-the-Art Portfolio Selection: Using Knowledge-based Systems to Enhance Investment Performance. Probus Publishing, 1992.

[36] Walter J., Ritter H. and Schulten K., "Nonlinear prediction with self organizing maps", Proc. International Joint Conference on Neural Networks, San Diego, June 1990, I 589 594

[37] Zhang X., Wong F. "A Decision Support Neural Network and Its Financial Applications " Technical Report, Institute of Systems Science, National University of Singapore, June, 1992.

[38] Zimmermann H., "Applications of the Boltzman machine in economics", in

Reider (Ed.), Methods of Operations Research, Vol 60/61, 14th Symposium on Operations Research, Verlag Anton Hain, 1990.

APPENDICES

CONOPT

General non-linear programming models with sparse non-linear constraints

The algorithm in CONOPT is based on the generalized reduced gradient (GRG) algorithm. All matrix operations are implemented by using sparse matrix techniques to allow very large models. Without compromising the reliability of the GRG approach, the overhead of the GRG algorithm is minimized by, for example, using dynamic feasibility tolerances, reusing Jacobians whenever possible, and using an efficient re-inversion routine. The algorithm uses many dynamically set tolerances and therefore runs, in most cases, with default parameters.

CONOPT is available as a subroutine library and as a subsystem under the modeling systems AIMMS, AMPL, GAMS, and LINGO. CONOPT is available for PCs and most workstations. All versions are distributed in compiled form. The system is continuously being updated, mainly to improve reliability and efficiency on large models. The latest additions are options for SLP and steepest edge.

GRG2

Non-linear programming

GRG2 uses an implementation of the generalized reduced gradient (GRG) algorithm. It seeks a feasible solution first (if one is not provided) and then retains feasibility as the objective is improved. It uses a robust implementation of the BFGS quasi-Newton algorithm as its default choice for determining a search direction. A limited-memory conjugate gradient method is also available, permitting solutions of problems with hundreds or thousands of variables. The problem Jacobian is stored and manipulated as a dense matrix, so the effective size limit is one to two hundred active constraints (excluding simple bounds on the variables, which are handled implicitly).

The GRG2 software may be used as a stand-alone system or called as a subroutine. The user is not required to supply code for first partial derivatives of problem functions; forward or central difference approximations may be used instead.

Documentation includes a 60-page user's guide, in-line documentation for the subroutine interface, and complete installation instructions.

GRG2 is written in ANSI FORTRAN. A C version is also available. Machine dependencies are relegated to the subroutine INITLZ, which defines three machine-dependent constants.

Lancelot

Unconstrained Optimization Problem

The LANCELOT package uses an augmented Lagrangian approach to handle all constraints other than simple bounds. The bounds are dealt with explicitly at the level of an outer-iteration sub problem, where a bound-constrained nonlinear optimization problem is approximately solved at each iteration.

The algorithm for solving the bounded problem combines a trust region approach adapted to handle the bound constraints, projected gradient techniques, and special data structures to exploit the (group partially separable) structure of the underlying problem.

The software additionally provides direct and iterative linear solvers (for Newton equations), a variety of preconditioning and scaling algorithms for more difficult problems, quasi-Newton and Newton methods, provision for analytical and finite-difference gradients, and an automatic decoder capable of reading problems expressed in Standard Input Format (SIF).

LANCELOT A is written is standard ANSI Fortran 77. Single- and double-precision versions are available. Machine dependencies are isolated and easily adaptable. Automatic installation procedures are available for DEC VMS, DEC ULTRIX, Sun UNIX, Cray UNICOS, IBM VM/CMS, and IBM AIX.

Mat-lab Optimization Toolbox

Linear programming, quadratic programming, unconstrained and constrained optimization of nonlinear functions, nonlinear equations, nonlinear least squares, minimax, multi objective optimization, semi-infinite programming.

Linear programming --- a variant of the simplex method. An initial phase is needed to identify a feasible point

Quadratic programming --- an active set method. A linear programming problem is solved to determine an initial feasible point.

Unconstrained minimization --- two routines are supplied. One implements a quasi-Newton algorithm, using either DFP or BFGS to update an approximate inverse Hessian, according to a switch selected by the user. Gradients may be supplied by the user; if they are not, finite differencing is used. The second routine uses the Nelder-Mead simplex algorithm, for which derivatives are not needed.

Constrained minimization --- sequential quadratic programming. The BFGS formula is used to maintain an approximation to the Hessian. Han's merit function is used to determine the step length at each iteration.

Nonlinear equations --- Newton's method and the Levenberg-Marquardt algorithm are supplied. The user chooses the algorithm by setting a switch.

Nonlinear least squares --- the Gauss-Newton method and the Levenberg-Marquardt method are supplied. The user makes the choice.

Minimax --- these problems can be formulated as constrained optimization problems, and a sequential quadratic programming algorithm is used to solve them here. Advantage is taken of the structure of the problem in the choice of approximate Hessian.

Multi objective optimization --- The problem is formulated as one of decreasing a number of objective functions below a certain threshold simultaneously, so it is viewed as a constrained optimization problem. Again, sequential quadratic programming is used to solve it.

Semi-infinite programming --- Cubic and quadratic interpolation is used to locate peaks in the infinite constraint set and therefore to reduce the problem to a constrained optimization problem.

Minos

Linear programming, unconstrained and constrained nonlinear optimization

Linear programming: A primal simplex method is used. A sparse LU factorization of the basis is maintained, using a Markowitz ordering scheme and Bartels-Golub updates as implemented in the LUSOL package of Gill, Murray, Saunders, and Wright.

Nonlinear objective, linear constraints: A reduced-gradient algorithm is used. This is an active-set method (a natural extension of the simplex method). The variables are classified as-, super basic, and non-basic, with the number of super basics indicating the effective non-linearity of the objective. The constraints are satisfied before the objective is evaluated. Feasibility is maintained thereafter. Search directions are generated using a quasi-Newton approximation to the reduced Hessian.

Nonlinear constraints: A projected augmented Lagrangian algorithm is used. As in Robinson's method, each major iteration solves a linearly constrained sub problem to generate a search direction. The sub problem objective is an augmented Lagrangian function. The sub problem constraints are the true linear constraints plus linearizations of the nonlinear constraints. Convergence is usually achieved, although the step length choice is heuristic. (A reliable merit function is not yet known.)

MINOS is designed to handle thousands of constraints and variables. Constraint data may be input from MPS files or via subroutine parameters. Non-linearities are specified by Fortran subroutines. (Ideally these should provide both functions and gradients. Missing gradients are estimated by finite differences.) The GAMS and AMPL systems may be used as alternative user interfaces. See their entries for details.

MINOS is distributed on floppy disk. Fortran 77 source code is provided, along with test problems and makes files for Unix, VMS and DOS systems.

Nag C Library

Linear programming, quadratic programming, minimization of a nonlinear function (unconstrained, bound-constrained, linearly constrained, and nonlinearly constrained), and minimization of a sum of squares.

For problems with nonlinear constraints, a sequential QP algorithm is used. For unconstrained problems and problems with simple bounds, quasi-Newton and conjugate gradient methods are provided. The Nelder-Mead simplex method is provided for unconstrained problems. For minimizing a sum of squares, a Gauss-Newton method is used. The LP and QP routines use a numerically stable active-set strategy.

An option-setting mechanism is provided in all routines, in order to keep the basic parameter-list to a minimum, while allowing a large degree of flexibility in controlling the algorithm. The routines have the ability to print the solution, as well as various amounts of intermediate output to monitor the computation.

Service routines are provided for checking user-supplied routines for first derivatives and for computing a covariance matrix for nonlinear least squares problems.

The NAG C Library is available in tested, compiled form for several hardware/software-computing environments.

Nag Fortran Library

Linear programming, mixed-integer linear programming, quadratic programming, minimization of a nonlinear function (unconstrained, bound constrained, linearly constrained, and nonlinearly constrained), and minimization of a sum of squares (unconstrained, bound constrained, linearly constrained, and nonlinearly constrained).

For problems with nonlinear constraints, a sequential QP algorithm is used. For unconstrained problems and problems with simple bounds, quasi-Newton, modified Newton, and conjugate gradient methods are provided. The Nelder-Mead simplex method is provided for unconstrained problems. For minimizing a sum of squares, a Gauss-Newton method is used. The LP and QP routines use a numerically stable active set strategy in which the linear constraint matrix may be dense or sparse. Problem data may be supplied in MPSX format.

An option-setting mechanism is provided in the more recent routines, in order to keep the basic parameter list to a minimum, while allowing a large degree of flexibility in controlling the algorithm. These routines also have the ability to print the solution, as well as various amounts of intermediate output to monitor the computation.

Service routines are provided for approximating first or second derivatives by finite differences, for checking user-supplied routines for first or second derivatives, and for computing a covariance matrix for nonlinear least squares problems.

The NAG Fortran Library is available in tested, compiled form for a large number of different hardware and software computing environments.

NLPQL

Smooth nonlinear programming with equality and inequality constraints

NLPQL solves smooth nonlinear programming problems, i.e. minimizes a nonlinear objective function subject to nonlinear equality and inequality constraints. It is assumed that all model functions are continuously differentiable.

The internal algorithm is a sequential quadratic programming (SQP) method. Proceeding from a quadratic approximation of the Lagrangian function and a linearization of the constraints, a quadratic sub problem is formulated and solved to get a search direction. Subsequently a line search is performed with respect to two alternative merit functions, and the Hessian approximation is updated by the modified BFGS formula.

Special features of NLPQL are

- Separate handling of upper and lower bounds on the variables,
- Reverse communication,
- Internal scaling,
- Initial multiplier and Hessian approximation,
- Feasibility with respect to bounds and linear constraints,
- Full documentation by initial comments.

NLPQL is written in double-precision Fortran 77 and organized in the form of a subroutine. Nonlinear problem functions and analytical gradients must be provided by the user within special subroutines or the calling program.

Optima Library

Unconstrained optimization, constrained optimization, sensitivity analysis

OPVM --- Unconstrained optimization, unstructured objective function; suitable for small problems.

OPVMB --- Optimization subject to simple bounds.

OPLS --- Unconstrained nonlinear least squares.

OPNL --- Nonlinear equations, by minimizing sum of squares of the residuals.

OPCG --- Nonlinear conjugate gradient method.

OPODEU --- Unconstrained optimization problems by tracing the solution curve of a system of ODEs (homotopy method).

OPTNHP --- Unconstrained optimization using the truncated Newton method; no Hessian storage or calculation required.

OPRQP --- Sequential quadratic programming, but superseded by the OPXRQP routine.

OPXRQP --- A more efficient implementation of sequential quadratic programming; uses the EQP variant.

OPSQP --- Another implementation of sequential quadratic programming, but uses the IQP variant, which gives rise to inequality-constrained sub problems.

OPALQP --- Similar to OPSQP, but uses an augmented Lagrangian line search function.

OPSMT --- Nonlinear programming, using a SUMT technique.

OPIPF --- Sequential minimization of a sequence of augmented Lagrangians.

OPODEC --- Homotopy method: traces the solution curve of a system of ODEs.

OPSEN --- Tests the sensitivity of the objective function around the optimal point of an unconstrained problem.

OPSEC --- Like OPSEN, but for the solution of a constrained problem.

Software is written in Fortran 77.

Optpack

Unconstrained optimization and nonlinear constrained optimization with special software to handle bound constraints, linear equality constraints, and general nonlinear constraints

Unconstrained optimization is performed using the conjugate gradient algorithm. Constrained optimization is performed using a new scheme that combines multiplier methods with preconditioning and linearization techniques to accelerate convergence.

The software is written in double precision Fortran. The code is documented by internal comments. Research reports providing the theoretical basis for the algorithms are available on request. User feedback is much appreciated.

Port

General minimization, nonlinear least squares, separable nonlinear least squares, linear inequalities, linear programming, and quadratic programming.

The nonlinear optimizers have unconstrained and bound-constrained variants, and use trust region algorithms. Gradients and Jacobians can be provided by the caller or approximated automatically by finite differences. The general minimization routines use either a quasi-Newton approximation to the Hessian matrix or a Hessian provided by the caller; the nonlinear least squares routines adaptively switch between the Gauss-Newton Hessian approximation and an "augmented" approximation that uses a quasi-Newton update. Function and, if necessary, gradient values may be provided either by subroutines or by reverse communication.

There is a special separable nonlinear least squares solver for the case of one nonlinear variable; it uses Brent's one-dimensional minimization algorithm for the nonlinear variable. Brent's algorithm is also available by itself, as is an implementation of the Nelder-Mead simplex method.

The feasible point (linear inequalities) and linear and quadratic programming routines start by taking steps through the interior and end with an active set strategy. The quadratic programming solvers use the Bunch-Kaufman factorization and thus can find local minimizes of indefinite problems.

None of the solvers is meant for large numbers of variables. When there are n variables and m equations (where m = 1 for general minimization), the nonlinear solvers require O(n ^ 2 m) or O(n ^ 3) arithmetic operations per iteration. The linear and quadratic solvers use dense-matrix techniques.

Software is written in ANSI Fortran 77, with single- and double-precision versions of all solvers. Machine-dependent constants are provided by subroutines I1MACH, R1MACH, and D1MACH.

PROC NLP (SAS/OR Software)

General and specialized nonlinear optimization

The NLP procedure offers a set of optimization techniques for minimizing or maximizing a continuous nonlinear function *f(x)* of n decision variables with boundary, general linear, and nonlinear equality and inequality constraints. PROC NLP supports a number of algorithms for solving this problem that take advantage of the special structure of the objective and constraint functions. Two algorithms are especially designed for quadratic optimization problems, and two other algorithms are provided for the efficient solution of nonlinear least-squares problems.

PROC NLP is part of SAS/OR Software, a fully integrated component of the SAS System. Along with its programming statements, PROC NLP uses SAS data sets (proprietary format) for input and for output. By taking advantage of the SAS System's Multiple Engine Architecture, PROC NLP can in effect read from and write to over fifty different database formats.

In addition to producing output SAS data sets, PROC NLP can print text output detailing the initial decision variable values, the search for an initial feasible solution, the optimization history, and the values of decision variables, derivatives, and covariance matrices at optimality.

Input Data

- Objective function and the constraints are specified using the programming statements of PROC NLP
- Additional data sets can be used to generate constraints and objectives
    - DATA= data set specifies an objective function that is a combination of n other functions
    - INQUAD= data set (sparse or dense format) specifies the objective of a quadratic programming problem
    - INEST= or INVAR= data set specifies initial values for the decision variables, the values of constants that are referred to in the program statements, as well as simple boundary and general linear constraints
    - MODEL= data set specifies a model saved from a previous execution of the NLP procedure

Output Data

- OUT= output data set contains variables generated in the program statements defining the objective function (and perhaps derivatives) plus any variables used in a DATA= input data set

- OUTEST= data set contains values of decision variables, derivatives, and covariance matrices at optimality, and can be used in subsequent PROC NLP calls as an INEST= input data set
- OUTMOD= data set contains the programming statements and can be used in subsequent PROC NLP calls as a MODEL= input data set

Optimizers

The following algorithms are available via PROC NLP for use with these categories of nonlinear programs:

- **Nonlinear min/maximization with linear constraints**
    - A trust-region algorithm (Dennis, Gay, & Welch, 1981, Gay, 1983, and Moré and Sorensen, 1983)
    - Two different Newton-Raphson algorithms using line search or ridging
    - Quasi-Newton algorithms updating either an approximation of the inverse Hessian or the Cholesky factor of an approximate Hessian
    - A double dogleg algorithm (Gay, 1983 and Dennis and Mei, 1979)
    - Various conjugate gradient algorithms with the Powell and Beale automatic restart update (Powell, 1977, and Beale, 1972), Fletcher-Reeves update, Poliak-Ribiere update, or conjugant-descent update (Fletcher, 1987)
    - The Nelder-Mead simplex algorithm with a modification of Powell's COBYLA implementation (Powell, 1992)
    - Any of the algorithms listed above, substituting the original Nelder-Mead simplex algorithm for the COBYLA version
- **Nonlinear min/maximization with nonlinear constraints**
    - A quasi-Newton algorithm that is a modification of Powell's Variable Metric Constrained Watch Dog (VMCWD) algorithm (Powell, 1978, 1982)

- The Nelder-Mead simplex algorithm with a modification of Powell's COBYLA implementation (Powell, 1992)

- **Nonlinear least squares with linear constraints**
  - The Levenberg-Marquardt algorithm (Moré, 1978)
  - a hybrid quasi-Newton algorithm (Fletcher & Xu, 1987, Lindstrom & Wedin, 1984, and Al-Baali & Fletcher, 1986)

- **Quadratic min/maximization with linear or boundary constraints**
  - Solve as a linear complementarily problem (if the symmetric matrix is positive/negative semi-definite for a min/maximization and the variables are restricted to be positive)
  - Use a general quadratic optimization active set algorithm (Gill, Murray, Saunders, & Wright, 1984)

Derivatives

PROC NLP may require derivatives of the objective function and the constraints. These can be obtained

- Analytically (using a special derivative compiler), the default method
- Via finite difference approximations
- Via user-supplied exact or approximate numerical functions

Problem Size Limitations

The size of a problem that PROC NLP can solve depends on the host platform, the available memory, and the available space for utility data sets. PROC NLP does not place any additional limits on problem size.

Available Platforms

The SAS System is supported on all major personal computer, workstation, and mainframe operating systems.

SQP

Nonlinear programming

SQP uses an implementation of Powell's successive quadratic programming algorithm and is aimed specifically at large, sparse nonlinear programs. It solves the quadratic programming sub-problems by using a sparsest-exploiting reduced gradient method. Sparse data structures are used for the constraint Jacobian, and there is an option to represent the approximate Hessian as a small set of vectors using a limited memory-updating scheme.

SQP requires the same user-supplied subroutines as GRG2 and has similar subroutine and data file interfaces. The entry describing GRG2 contains more details.

SQP is written in ANSI FORTRAN. Machine dependencies are relegated to the subroutine INITLZ, which defines three machine-dependent constants.