1999

AN INVESTIGATION OF EVOLUTIONARY COMPUTING IN SYSTEMS IDENTIFICATION FOR PRELIMINARY DESIGN

WATSON, ANDREW HARRY

http://hdl.handle.net/10026.1/1669

http://dx.doi.org/10.24382/1403
University of Plymouth

An Investigation Of Evolutionary Computing In Systems Identification For Preliminary Design

Ph.D. Thesis submitted by A.H.Watson

1999

AN INVESTIGATION OF EVOLUTIONARY COMPUTING IN SYSTEMS IDENTIFICATION FOR PRELIMINARY DESIGN

by

ANDREW HARRY WATSON

A thesis submitted to the University of Plymouth
in partial fulfillment for the degree of

DOCTOR OF PHILOSOPHY

School of Computing
Faculty of Technology

In collaboration with
Rolls Royce Plc.

March 1999

Copyright Statement

Signed .....

(ANDREW HARRY WATSON)

Date 30/7/99

An Investigation Of Evolutionary Computing In Systems Identification For Preliminary Design

Andrew Harry Watson

ABSTRACT

This research investigates the integration of evolutionary techniques for symbolic regression. In particular the genetic programming paradigm is used together with other evolutionary computational techniques to develop novel approaches to the improvement of areas of simple preliminary design software using empirical data sets. It is shown that within this problem domain, conventional genetic programming suffers from several limitations, which are overcome by the introduction of an improved genetic programming strategy based on node complexity values, and utilising a steady state algorithm with sub-populations. A further extension to the new technique is introduced which incorporates a genetic algorithm to aid the search within continuous problem spaces, increasing the robustness of the new method. The work presented here represents an advance in the field of genetic programming for symbolic regression with significant improvements over the conventional genetic programming approach. Such improvement is illustrated by extensive experimentation utilising both simple test functions and real-world design examples.

ACKNOWLEDGMENTS

Many thanks to Dr. Ian Parmee for his guidance and motivation throughout this research. I am also grateful to Graham Purchase from Rolls Royce plc. and the EPSRC (UK) who jointly funded the research.

To Jo, Bryony and Beth.

AUTHORS DECLARATION

At no time during the registration for the degree of Doctor of Philosophy has the Author been registered for any other University award.

This study was financed with the aid of an EPSRC/RR CASE studentship.

Industrial Collaborator : Rolls Royce Plc.

The research has been presented by the author at 7 relevant international technical conferences. Rolls Royce Plc. has been visited regularly for technical discussions and consultation purposes.

PUBLICATIONS:

Watson, A. H.; Parmee, I. C. Systems Identification Using Genetic Programming. Proceedings of ACEDC: p248-255. University of Plymouth. 1996.

Watson, A. H.; Parmee, I. C. Identification Of Fluid Systems Using Genetic Programming. Proceedings of the 2nd Online Workshop on Evolutionary Computation. p45-48. On the Inernet WWW served by Nagoya University. 1996.

Watson, A. H.; Parmee, I. C. Identification Of Fluid Systems Using Genetic Programming. Proceedings EUFIT '96. Vol I, p395-399. ELITE Foundation. 1996.

Watson, A.H. ; Parmee, I. C. Steady State Genetic Programming with Constrained Complexity Crossover. Procs. 2nd Annual Conference Genetic Programming (GP'97) MIT Press, 1997.

Watson, A.H. ; Parmee, I. C. Steady State Genetic Programming with Constrained Complexity Crossover Using Species Sub-Populations. Procs. 7th International Conference on Genetic Algorithms. (ICGA 97). Morgan Kaufmann. 1997.

Watson, A. H.; Parmee, I. C. An Improved Genetic Programming Strategy For Preliminary Design Model Development. Proceedings EUFIT '97. Vol I, p682-686. ELITE Foundation. 1997.

Watson, A. H.; Parmee, I. C. Improving Engineering Design Models Using An Alternative Genetic Programming Approach. Proceedings of ACDM, p193-206. Springer. 1998.

PRESENTATIONS AND CONFERENCES ATTENDED:

ACEDC '96 2nd International Conference On Adaptive Computing In Engineering Design and Control. University of Plymouth, UK March 1996.

WEC '96 2nd Online Workshop On Evolutionary Computation. WWW Served by Nagoya University, March 4-22 1996.

EUFIT '96 4th European Congress on Intelligent Techniques and Soft Computing. Aachen, Germany, September 2-5 1996.

GP '97 PhD Student Workshop. Stanford University, San Francisco, USA, July 12th 1997.

GP '97 2nd Annual Conference on Genetic Programming. Stanford University, San Fransisco, USA, July 13th-16th 1997.

ICGA '97 7th International Conference on Genetic Algorithms. Michigan State University, Michigan, USA, July19th-23rd 1997.

EUFIT '97 5th European Congress on Intelligent Techniques & Soft Computing. Aachen, Germany, September 8-11 1997.

ACDM '98 3rd International conference on Adaptive Computing In engineering Design and Manufacture. University of Plymouth, UK April 1998.

Director of Studies:

Dr I. C. Parmee,
Director,
Pltmouth Engineering Design Centre,
University of Plymouth.

Industrial Supervisor:

Graham Purchase
Chief Design Engineer,
Turbines,
Rolls Royce, plc.
Filton
Bristol.

Signed .....

Date .....

LIST OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

During the last thirty years there has been a growing interest in computer based problem solving systems based on principles of evolution. This approach, known collectively as Evolutionary Computation (EC), includes genetic algorithms (GA's), genetic programming (GP), evolutionary programming (EP) and evolution strategies (ES). When applied to practical problem solving, all begin with a population of contending trial solutions to the task at hand. New solutions are created by randomly altering the existing solutions. An objective measure of performance is used to assess the "fitness" or "error" of each trial solution, and a selection mechanism determines which solutions should be maintained as "parents" for the subsequent generation. The differences between the procedures are characterised by the types of alterations that are imposed on solutions to create offspring, the methods employed for selecting new parents, and the data structures that are used to represent solutions. These techniques are now being used extensively, for instance, in the fields of design, pattern recognition, engineering, control, scheduling, and systems identification.

The objective of the research described within this thesis is to develop evolutionary strategies for the identification of improved mathematical representations relating to areas of preliminary design software which contain a high degree of approximation.

In general, preliminary engineering design practice involves look-up tables or graphs based upon empirical data that is not easily represented computationally. During preliminary design, approximate solutions can provide sufficient guidance for the engineer to determine optimal design directions. By using approximate functions to describe the physical process

during preliminary design the engineer is able to rapidly investigate many possible design solutions before progressing to more definitive analysis techniques such as Computational Fluid Dynamics (CFD) and Finite Element Analysis (FEA). A contributing factor to function approximation may be the inclusion of empirically derived coefficients (i.e. discharge, drag, etc.).

The identification, manipulation and optimisation of these approximate functions that describe the physical process will be investigated using genetic programming and by the development of complementary evolutionary computation and adaptive search techniques.

1.1 The Engineering Design Process

There is no single universally accepted sequence of steps that leads to a workable design. Design is a sequential process consisting of many design operations. Examples of the operations might be:-

- Exploring the alternative systems that could satisfy the specified need.
- Formulating a mathematical model of the best system concept.
- Specifying specific parts to construct a component of a sub-system.
- Selecting a material from which to manufacture a part.

Each operation requires information, some of it is provided in general technical and business information, but some is very specific information that is needed to produce a successful outcome. Acquisition of information is a vital and often very difficult step in the design process, but fortunately it is a step that usually becomes easier with time (this process is called experience).

Once armed with the necessary information, the design engineer (or design team) carries out the design operation by using the appropriate technical knowledge and computational and/or experimental tools. The typical design project will break itself down into a number of time phases which are listed below (Pahl & Beitz, 1988).

Phase I - Feasibility Study

The purpose of the feasibility study is to initiate the design and establish the line of thinking. The goal in this phase is to validate the need, produce a number of possible solutions, and evaluate the solutions on the basis of physical feasibility, economic viability, and financial feasibility. This stage sometimes is called conceptual design.

Phase II - Preliminary Design

Starting with a set of useful solutions developed in phase I, the goal of the preliminary design is to quantify the parameters so as to establish the optimal solution. A preliminary design usually is concerned only with order-of-magnitude estimates of design performance and cost. At this stage it may be necessary to construct a mathematical model and conduct a simulation of the component's performance on a digital computer.

Phase III - Detailed Design

The purpose of the detailed-design phase is to develop a complete engineering description of a tested and producible design. The process involves complex, computationally expensive models and calculations as well as expensive testing of components.

The phases that follow the first three listed include planning for manufacture, distribution, use and retirement of the product and do not concern the area of study related to this thesis.

The preliminary design phase often utilises data presented in a graphical form, which is used to calculate various design specific goals. If optimisation of the design is to be undertaken by a computer, the data needs to be either directly inputted (then interpolated), or a mathematical equation representing the data can be used. The identification of the mathematical model that best represents the data is the subject for this thesis.

1.2 The Role Of Models In Engineering Design

A model is an idealisation of a real-world situation that supports the analysis of a problem. A model may be either descriptive or prescriptive. A *descriptive model* helps to understand a real-world system or phenomenon; an example is a cutaway model of an aircraft gas turbine. Such a model serves as advice for communicating ideas and information. However, it does not help to predict the behaviour of the system. A *predictive model* is used primarily in engineering design because it helps to both understand and predict the performance of the system.

Models can be classified as follows:

- Static-dynamic
- Deterministic-Probabilistic, and
- Iconic-analogue-symbolic

A ***static model*** is one whose properties do not change with time; a model in which time-varying effects are considered is ***dynamic***. In the deterministic-probabilistic class of models there is differentiation between models that predict what will happen. A ***deterministic model*** describes the behaviour of a system in which the outcome of an event occurs with certainty. In many real-world situations the outcome of an event is not known with certainty, and these must be treated with ***probabilistic models***. An ***iconic model*** is one that

represents the physical characteristics of the system being modelled. Examples are a scale model of an aircraft for wind tunnel test and an enlarged model of a polymer molecule. Iconic models are used primarily to describe the static characteristics of a system, and they are used to represent entities rather than phenomena. ***Analogue models*** are those that behave like the real systems. They are often used to compare something that is unfamiliar with something that is familiar. Unlike an iconic model, an analogue model need look nothing like the real system it represents. It must either obey the same physical principles as the physical system or simulate the behaviour of the system. An ordinary graph is an analogue model because distances represent the magnitude of the physical quantities plotted on each axis. Since the graph describes the real functional relation that exists between those quantities, it can be seen as a model. ***Symbolic models*** are abstractions of the important quantifiable components of a physical system. A mathematical equation expressing the dependence of the system output parameter on the input parameters is a common symbolic model. A symbol is a shorthand label for a class of objects, a specific object or a state of nature, or simply a number. Symbols are useful because they are convenient, add to simplicity of explanation, and increase the generality of the situation. A symbolic model probably is the most important class of model because it provides the greatest generality in attacking a problem. The use of a symbolic model to solve a problem leads to quantitative results. Further distinction can be made between symbolic models that are theoretical, based on established and universally accepted laws of nature, and empirical models, which are the best approximate mathematical representations based on existing experimental data.

The solution of models by the straightforward application of mathematical techniques has been the classical approach, but only the simplest (and hence usually most unrealistic) models can be solved with classical analytic methods. The widespread use of the digital computer has greatly expanded the scope and usefulness of mathematical modelling. The use of numerical methods for solution and the ease with which iterative and evolutionary

procedures can test many specific states of the model have established evolutionary computer modelling as a powerful tool of engineering design.

1.3 Evolutionary Computation History

Genetic algorithms, evolutionary programming and evolutionary strategies were developed essentially in parallel, in the 1960's and 1970's. Genetic algorithm and evolutionary programming strategies have been primarily developed in the United States, whereas evolution strategies originated in Germany. Genetic programming is the most recent addition to the field of evolutionary computation and was developed in its current form in the late 1980's in the United States.

The development of genetic algorithms began in the 1950's through the use of computers by biologists to simulate natural genetic systems. One of those doing work most closely related to the current concepts of genetic algorithms was A.S. Fraser, who began publishing in the field in the late 1950s (Fraser, 1957). Fraser worked in the area of epistasis (suppression of the effect of a gene) and represented each of three parameters of an epistatic function as five bits in a 15-bit string. Fraser was working with natural systems, and while his work resembled function optimisation currently being solved by genetic algorithms, he did not consider the possibility of applying his methodology to artificial systems (Fraser, 1960, 1962).

John H. Holland of the University of Michigan was also beginning to publish in the early 1960s. Holland, together with his students has probably had more influence on the field of genetic algorithms than any others.

Holland's interest in machine intelligence led to the development and application of the capabilities of genetic algorithms to artificial systems. He taught courses in *adaptive systems* in the early 1960s while laying the groundwork for applications to artificial systems with his publications on adaptive systems theory (Holland, 1962). Holland's systems were adaptive because of their robustness in spite of changes and uncertainty in the environment. Further, they were *self-adaptive* in that they could make adjustments based on their interaction with the environment over time.

One of Holland's many contributions was his use of a population of individuals (chromosomes) in the search process, rather than the use of only a single individual as was common at the time. He also derived the schema theorem, which shows that schema (fundamental building blocks of individual chromosomes) that are more "fit" with respect to a defined fitness function are more likely to reproduce in successive generations of the population of chromosomes.

Beginning in the 1960s Holland's students routinely used reproduction, crossover, and mutation in their applications. Several of Holland's students made significant contributions to the genetic algorithm field, often starting with their Ph.D. dissertations, including K.A. De Jong, D.E. Goldberg and J. Koza..

The term "genetic algorithm" was used first by Bagley (Bagley, 1967) in his dissertation, which utilised genetic algorithms to find parameter sets in evaluation functions for game playing.

In 1975 Holland published one of the field's most important books, entitled *Adaptation in Natural and Artificial Systems* (Holland, 1975). Also in 1975, K. A. De Jong, one of Holland's students, published his Ph.D dissertation entitled, "An analysis of the Behaviour of a Class of Genetic Adaptive Systems". As part of his dissertation, De Jong put forward a

set of five test functions designed to measure the convergence of the algorithm. Two metrics were devised, one to measure the convergence of the algorithm, the other to measure the ongoing performance. De Jong examined the effects of varying four parameters (population size, crossover probability, mutation probability, and generation gap) on the performance of six main kinds of genetic algorithm paradigms (De Jong, 1975). De Jong's five-function test bed and two performance metrics still provide some of the most commonly referenced genetic algorithm performance criteria. David E. Goldberg was another of Holland's students. He has concentrated on engineering applications of genetic algorithms. He is a former gas pipeline engineer; his Ph.D. dissertation considered a 10-compressor, 10-pipe, steady-state, serial gas pipeline problem (Goldberg, 1983). The goal was to provide a strategy that minimised the power consumption in the pumping stations, subject to pressure-related constraints. In 1989, Goldberg published one of the most important books on genetic algorithms, *Genetic Algorithms in Search, Optimisation and Machine Learning* (Goldberg, 1989).

In the United States, Larry J. Fogel and his colleagues developed what they named Evolutionary programming. Evolutionary programming uses the selection of the fittest, similar to genetic algorithms, but the only structure-modifying operation allowed is mutation. Fogel and his colleagues mainly worked with finite state machines and were interested in machine intelligence. They were able to solve a problem involving significant epistasis that was quite difficult for genetic algorithms. Fogel (Fogel, 1994) has described evolutionary programming as taking a fundamentally different approach than genetic algorithms. Genetic algorithms are described as a bottom up process of adaptive genetics but evolutionary programming acts as a top-down process of adaptive behaviour. Fogel summarises evolutionary programming as implementing "survival of the more skilful" rather than "survival of the fittest" emphasised by genetic algorithm developers.

In Germany, at the Technical University of Berlin, I. Rechenberg developed what he called Evolutionstrategie (evolution strategies) during the mid-1960s. He was working on engineering optimisation problems that involved airfoil design, including physical configurations of a series of hinged plates in a wind tunnel. He used evolution strategies to vary the angle of the plates and of the tube. Rechenberg and his student, H. P. Schwefel, used the first computer at the university to simulate various versions of the strategy (Rechenberg, 1965; Schwefel, 1965). In the early 1970s, Rechenberg published a book that is considered the foundation for this approach (Rechenberg, 1973).

The fourth major area of evolutionary computation is genetic programming. Some of the earliest related work was completed by Friedberg and other co-workers (Friedberg, 1958; Friedberg et al., 1959). They worked with fixed-length computer programs that were coded by another program designed to optimise the performance of the fixed-length program. Their programs each comprised a set of 64 instruction, each instruction being 14 bits long. The programs were defined such that every arrangement of the 14 bits was a valid instruction, and each set of 64 instructions was a valid program. Unfortunately, the results of the efforts did not live up to expectations. In retrospect, there were probably two main reasons for this. First, the programs were limited in length to 64 instructions: a "failure" was returned if the program did not terminate successfully by the end of the 64th instruction (even if there was a loop). Second, there was only one program; thus there was a population of just one that evolved.

The two limitations just cited were successfully dealt with by John Koza who developed genetic programming (in its current form) in the late 1980's. Working at Stanford University, Koza's system is designed to evolve computer programs genetically using a population of tree-shaped chromosomes (Koza, 1992). The origins of GP can be traced back to earlier work (Cramer, 1985) who used evolutionary techniques for program

induction, and another paper (Bickel & Bickel, 1987) which used genetic methods to if-then expert systems which incorporated tree structured rules. Various other people have been credited with important earlier papers including the use of genetic operators for program induction (Fujiki, 1986), the induction of if-then clauses for game strategies (Fujiki & Dickinson, 1989) and the application of genetic algorithms to automatic program generation (Hicklin, 1986).

Many seemingly different problems in artificial intelligence, symbolic processing, and machine learning can be viewed as a search for a computer program that produces some desired output for particular inputs. A computer program being a collection of instructions which when executed perform a specific task. When viewed in this way, the process of solving these problems becomes equivalent to searching a space of possible computer programs for the fittest individual computer program. The search space is the space of all possible computer programs composed of functions and terminals appropriate to the problem domain and genetic programming provides an effective way to search for them.

One problem that arises when using computers to solve problems is that existing methods of machine learning, artificial intelligence, self-improving systems, self-organising systems, neural networks, and induction do not seek solutions in the form of computer programs. Instead, existing paradigms involve specialised structures (e.g., weight vectors for neural networks, formal grammars, coefficients for polynomials, production rules and chromosome strings in the conventional genetic algorithm). Each of these specialised structures can facilitate the solution of certain problems, and many of them facilitate mathematical analysis that might not otherwise be possible. If computers are to be used to solve problems without being explicitly programmed, i.e. not including partial solutions to the problem within the problem solving technique, then a very good candidate for the structures required are *computer programs*.

They offer the flexibility to: -

- Perform operations in a hierarchical way.
- Perform alternative computations conditioned on the outcome of intermediate calculations.
- Perform iterations and recursions.
- Perform computations on variables of many different types.
- Define intermediate values and subprograms so that they can be subsequently reused.

The size and shape of the structures need not be specified in advance, as is generally the case in a genetic algorithm. These attributes of the solution should emerge during the problem-solving process as a result of the demands of the problem. The size, shape, and structural complexity should be part of the answer produced by the problem solving technique not part of the question when used to solve symbolic regression problems. An immediate problem is how to find the desired program in the space of possible programs. The space of possible computer programs is too vast for a blind random search. Thus there is a need to search in some adaptive and intelligent way.

1.4 Current Research In Evolutionary Computing

A genetic algorithm operates by repeatedly modifying a population of artificial structures through the application of genetic operators. GA's use fitness information exclusively; they do not require gradient information or other internal knowledge of the problem. A genetic algorithm's data structure consists of one or more *chromosomes,* which may be represented as a string of bits, so the term string is often used. Other possible representations include real number encoding (Goldberg, 1991 (a)), structured GA's (Dasgupta, 1991, 1992), and high level computer programs with variable-length strings (Koza, 1992).

Theoretical research in GA's covers the modelling and analysis of GA's using Markov chains and other statistical methods; the search for optimal control-parameter setting; the design of problem representations and genetic operators; the construction and solution of difficult problems; the design of mechanisms for niching and for the maintenance of population diversity; the parallel implementation of GA's; the design of hybrid GA's (Davis, 1991) that incorporate ideas borrowed from neural networks, simulated annealing, fuzzy logic, hill-climbing, and tabu search; and the comparison of algorithms. Applied research has covered problems in classification, combinatorial optimisation, design, function optimisation, information retrieval, machine learning, noise-tolerant problem solving, scheduling, search, simulation, and structural optimisation.

Theoretical work in genetic programming covers schema theory (Koza, 1992), (Poli & Langdon, 1997; Haynes, 1997). The investigation of the crossover operator (Angeline, 1997), and the effect of non-coded segments or introns (Andre & Teller, 1996; Banzhaf et. al. 1997). Investigation of various mutation methods has been investigated by Chellapilla (Chellapilla, 1997) where crossover is not used within the genetic programming paradigm. The effect of code growth in genetic programming has also been investigated by various people including Soule (Soule et. al. 1996), and Langdon (Langdon, 1997). Work on reuse of sub-trees within individuals has been investigated (Koza, 1994) where Automatically Defined Functions (ADF's) pursue the general goal of promoting modularity within the solution to the problem at hand.

1.5 Current Research In Evolutionary Systems Identification

Research concerning systems identification using GA's includes effects of control parameters for on-line performance of genetic algorithms for function optimisation (Schaffer et. al. 1989), (Messa, 1992), (Johnson & Husbands, 1991) and (Goldberg &

Richardson, 1987), the use of structured genetic algorithms (Dasgupta, 1991) and structured genetic algorithms for system identification (Iba et. al. 1993), using evolving polynomial networks (Kargupta & Smith, 1992) and the use of messy GA's (Goldberg, et. al. 1991(b)), and (Goldberg, et. al. 1993).

Research using genetic programming for systems identification include combined regression algorithms with genetic programming (Jiang, 1992 & 1993) and (Jiang & Wright, 1992). Hitoshi Iba et. al. have published papers on solving system identification (symbolic regression) problems using genetic programming (Iba, et. al. 1996(a)) and has also published a paper on random tree generation (Iba, 1996(b)).

1.6 Research Objectives

The objectives of the research can be summarised as follows:-

- To identify the utility of evolutionary computation and in particular genetic programming for systems identification.
- To develop appropriate evolutionary strategies for systems identification.
- The integration of complementary adaptive search and traditional optimisation techniques for systems identification..
- The improvement of areas of simple engineering software using the developed strategies.

1.7 Thesis Overview

This chapter has outlined the engineering design process and in particular the role of approximate mathematical models within preliminary design. A review of current research in

the field of evolutionary computation suggests that the genetic programming paradigm is the best-suited evolutionary computation algorithm for use within this field. Existing methods of regression and systems identification are presented in Chapter 2. This chapter discusses the possible ways in which an engineer can formulate a mathematical model using empirical data. This includes the genetic algorithm, neural networks, and genetic programming for simple systems identification problems and discusses advantages and disadvantages of these techniques together with the arguments for using the genetic programming paradigm. Chapter 3 explains the conventional genetic programming paradigm (Koza, 1992) in detail and includes recent work on aspects of genetic programming. Chapter 4 presents a comparison of techniques for solutions to various problems, which are also used to develop complimentary search techniques, before attempting to model simple engineering systems. Chapter 5 addresses the problems encountered with using standard genetic programming and introduces improved genetic programming methods for systems identification. The new technique is tested on various problems and includes various run parameter sets which show how the new genetic programming approach can be used depending on the problem being solved. Chapter 6 applies these new techniques to the simple engineering systems first presented in Chapter 4.

The final chapter, Chapter 7, provides a detailed discussion on the results presented in the previous chapters and the techniques developed within this thesis. The chapter also presents the conclusions from the research and possible future research directions.

CHAPTER 2

REGRESSION TECHNIQUES

The following chapter describes the main regression techniques available to the engineer and also introduces evolutionary techniques as a viable alternative to standard regression techniques. The advantages and disadvantages of these methods are discussed and a case for using the genetic programming paradigm is presented.

The mathematical modelling of any system requires the collection of relevant data. Once the data has been obtained, usually through empirical experimentation, a search for a mathematical formula which can best describes the data can commence. This process of finding a mathematical relationship from the data is termed regression analysis.

Regression is defined as the analysis or measure of the association between a *Dependant Variable* and one or more *Independent Variables* (Borowski & Borwein, 1989). Thus regression is concerned with the nature of association between variables. If a law exists connecting the variables, the nature of the association is stated as a mathematical equation. The equation can then be used to predict values of one variable for given values of the other variables.

Regression is the traditional approach to empirical modelling. The regression problem is formulated in such a way that the regression of a dependent variable y on an independent variable x is the computation of the most probable value of y for each value of x based on a finite number of possibly noisy measurements of x and the associated values of y. The values of the parameters are chosen to make the best fit to the observed data. In the case of linear regression, for example, the functional form is:-

$$y = ax + b, \tag{2.1}$$

Here the dependent variable y is assumed to be a linear function of independent variable x. The unknown parameters a and b are the linear coefficients.

Regression models are classified in a number of ways. For example, models are classified into *linear* or *non-linear regression* models according to the complexity of the functional form. Regression models are also classified in terms of the number of independent variables i.e. *univariate* or *multivariate.*

2.1 Empirical Modelling Methods

Empirical models are constructed based on a set of experimental data or observations of a process. The problem of developing an empirical model for a process can be viewed as the problem of approximating an input-output function mapping from a given set of experimental data.

Historically, relationships were established solely by examining the data - a difficult task if the relationship is complex, multi-variable, or if a high level of noise exists, due to experimental error, as often occurs in real-world problems. Moreover, the examination is easily influenced by the individuals desires and expectations. Statistical methods were among the first tools developed to help a researcher find the relationships between observed facts. Statistical methods are often based on the following assumptions:

- The data is normally distributed.
- The equation relating the data is of a specific form, for example, linear, quadratic, or a specified polynomial.
- The variables are independent.
- There is sufficient data to perform the statistical analysis.

If the problem meets the required assumptions, statistical methods represent a valuable tool for providing solutions. However real-world problems seldom meet these criteria.

2.2 Parametric and Non-Parametric Regression

The third way of classification of regression models concerns *parametric* or *non-parametric regression* according to the interpretation of the unknown parameters. *Parametric regression* model usually refers to the regression model where the form of the functional relationship is known (e.g., linear regression or a specified polynomial regression). The functional form contains some (usually small) number of unknown (but well defined) parameters whose values can be computed from the best fitting of the data. Typically, the unknown parameters of a parametric model have meaningful interpretation. The simplest example is the univariate *linear regression* model in the form:-

$$y = B_0 x + B_1 \qquad (2.2)$$

This is one kind of parametric regression because the function form of the dependence of y on x is specified, even though the value of the parameters B_0 and B_1 are not. The linear regression model makes several assumptions about the data, including linearity of the function of the explanatory variables, independence of the random errors, and equality of the variances of the random errors. Parametric regression therefore concerns the formulation of an equation containing independent variables and related coefficients. This is commonly formulated as an equation in which the independent variables have parametric coefficients and is therefore termed parametric regression.

On the other hand, *non-parametric regression* does not need to specify the form of the unknown functional relationship. No a priori knowledge about the form of the unknown function may be available. The function is still modelled using an equation containing unknown parameters but in a way which allows the class of functions which the model can represent to be very broad. Typically the equation, in some functional form, has many

unknown parameters, and none of the parameters have any physical meaning in relation to the problem to be solved. Neural networks can be used as non-parametric regression models.

The most commonly used regression method is the method of *least squares* where the sum of the squares of the differences between the observed and the theoretical values is minimised. This form of parametric regression can be linear, or by increasing the number of parameters, *polynomial regression* can be used. While this approach has been widely used, it suffers from a few drawbacks. Polynomial functions are not very flexible since they have all orders of derivatives everywhere, a seventh order polynomial function includes powers of x from seven down to zero and so the functional structure is set and cannot be changed. Individual observations can also have a huge influence on remote parts of the curve, if that particular data point is removed then the resulting polynomial equation is significantly changed. There are several ways to repair the drawbacks of polynomial fitting. One is to allow possible discontinuities of derivative curves. This leads to the *spline approach.*

2.3 Cubic Splines

One of the most popular methods for accurately drawing smooth curves through a series of points is the cubic spline (Lancaster & Salkauskas, 1986, and Cox, 1990). In theory, given a series of N points, an equation involving an N-1 degree polynomial can be devised. When the polynomial function is drawn as a graph, it passes through each of the points. This produces N linear equations which can be solved. The performance of the technique suffers with large numbers of test points. Firstly, the Gaussian elimination part of the method slows down markedly as it is asked to handle more unknowns also the resulting polynomials can be difficult to compute. For example, a polynomial equation that can thread its way through 100 points will contain a sub-expression x^{99}. Even for small values of x an attempt to calculate this will cause an arithmetic overflow. There are alternatives that allow the degree

of the polynomials involved to be minimised by dividing the data into smaller data sets. Low order polynomial curves are then used to fit each data set and a function is used to ensure a smooth transition between each curve. The resulting interpolation or smoothing function is called a *piecewise polynomial function*. The most widely used of these functions is the cubic spline.

Cubic splines dispense with the one large N-1 degree polynomial that goes through all of the points, replacing it with a number of simpler cubic polynomials that are individually responsible for drawing a line only between adjacent pairs of points. The cubic equations require 4 variables each, these variables, a_0, a_1, ... , a_3, are defined within a cubic equation thus:-

$$y = a_0x^3+a_1x^2+a_2x+a_3 \qquad (2.3)$$

If for example 5 data points are to be fitted using cubic splines, 4 cubic equations are required to join all points, and a total of 16 unknowns are required to represent the data. Each cubic polynomial must pass through two points, its start and end point. This provides two equations for each polynomial and 8 equations in all. However 16 equations are required to solve for the 16 unknowns, therefore extra conditions are required. Firstly the slope at the end of a segment should be the same as the slope of the next line at its start, this provides a further three equations, giving 11 equations in total. Further constraint for the problem is achieved by attempting to make the join between adjacent segments even smoother, i.e. by demanding that the second derivatives are also equal. This provides another 3 equations, 14 in total, and the final 2 equations are obtained by stating that the first and last points have a second derivative equal to zero.

The method is very prone to noise, i.e. inaccuracies in the data will dramatically change the curve. Any change in the data, possibly by the inclusion of an extra dimension, will require all of the cubic splines to be recalculated. No representative mathematical function is produced which will describe all of the data in a usable way. The method produces a series of cubic equations which when joined together, fit the data given. These 'piecewise' functions will not produce a single equation, which best represents the data given and so cannot be used for systems identification. However it is a popular technique, used mainly in the field of computer aided design and computer graphics.

2.4 Surface Fitting Using Polynomials

Polynomials also play a major role in surface fitting where additional dimensions significantly increase complexity. Problems related to existing curve fitting techniques become more acute, and complex mathematical analysis is required to produce good results.

The method must now have to cope with polynomials in *two* variables. The surface under investigation has to be represented as a series of 'patches' of the polynomial functions, normally bi-cubic patches (Lancaster & Salkauskas, 1986), and this leads to the formation of a large matrix which requires solving at increased computational expense. Again, as with spline fitting, no useful information is derived about the system under investigation due to the piecewise polynomial functions, or patches, used and so these surface fitting techniques cannot be used for symbolic regression problems.

2.5 Symbolic Regression

Symbolic regression (or function identification) involves finding a mathematical expression, *in symbolic form*, that provides a good, best, or perfect fit between a given finite sampling of values of the independent variables and the associated values of the dependent variables (Koza, 1992). That is, symbolic regression involves finding a model that fits a given sample of data. When the variables are real-valued, symbolic regression involves finding *both* the functional form and the numeric coefficients for the model.

This approach is also called nonparametric regression, the aim of which is to relax assumptions on the form of a regression function, and to let data search for a suitable function that adequately describes the available data. In the case of noisy data from the real world, this problem of finding the model from the data is often called *empirical discovery*. These approaches are powerful in exploring fine structural relationships and provide very useful diagnostic tools for parametric models.

2.6 Computer Intelligence

Computer intelligence involves computational techniques that exhibit an ability to learn and/or adapt to new situations. Computational intelligence systems are often designed to mimic one or more aspects of biological intelligence. These methods can be used to evolve solutions to regression problems and an overview of current methods is presented.

2.6.1 The Genetic Algorithm

Genetic algorithms mimic some of the processes observed in natural evolution. Biologists have been intrigued with the mechanics of evolution since the evolutionary theory of biological change gained acceptance through the work of Darwin in the mid 19th century (Darwin, 1859). Evolution takes place on *chromosomes* - organic devices for encoding the

structure of living beings. A living being is created partly through a process of decoding chromosomes. The specifics of chromosomal encoding and decoding processes are not fully understood, but some general, widely accepted features of the theory are:

- Evolution is a process that operates on chromosomes rather than on the living beings they encode.
- Natural selection is the link between chromosomes and the performance of their decoded structures. Processes of natural selection cause these chromosomes that encode successful structures to reproduce more often than those that do not.
- The process of reproduction is the point at which evolution takes place. Mutations may cause the chromosomes of biological children to be different from those of their biological parents, and recombination processes may create quite different chromosomes in the children by combining material from the chromosomes of two parents.
- Biological evolution has no memory. Whatever it knows about producing individuals that will function well in their environment is contained in the gene pool - the set of chromosomes carried by the current individuals - and in the structures of the chromosome decoders.

These features of natural evolution intrigued John Holland in the early 1970's (Holland, 1975). Holland believed that, appropriately incorporated in a computer algorithm, they might yield a technique for solving difficult problems in a similar manner to nature i.e. through evolution. He began investigating algorithms that manipulate strings of binary digits analogous to *chromosomes*. Holland's algorithms carried out simulated evolution on populations of such chromosomes. Like nature, his algorithms solved the problem of finding good chromosomes by manipulating the material in the chromosomes. Like nature, they knew nothing about the type of problem they were solving. The only information they were given was an evaluation of each chromosome they produced, and their only use of that

evaluation was to bias the selection of chromosomes so that those with the best evaluations tended to reproduce more often than those with bad evaluations.

These algorithms, using simple encoding and reproduction mechanisms, solved some extremely difficult problems. Like nature, they did so without knowledge of the decoded world. They were simple manipulators of simple chromosomes. Yet when the descendants of those algorithms are used today, it is found that they can evolve better designs, find better schedules, and produce better solutions to a variety of other important problems that cannot be solved as well using other techniques.

Before discussing in detail the simple genetic algorithm certain terminology used by researchers who work with genetic algorithms has to be mastered. Because genetic algorithms are rooted in both natural genetics and computer science, the terminology used in the GA literature is a mix of the natural and the artificial. The *strings* or *individuals* of artificial genetic systems are analogous to *chromosomes* in biological systems. In natural terminology, chromosomes are composed of *genes*, which may take on some number of values called *alleles*. In its simplest form, the GA consists of five basic steps - initialisation, evaluation, selection, crossover and mutation. The iterative sequence of selection, crossover, mutation and evaluation is known as a generation. Figure 2.1 shows the structure of the simple Genetic Algorithm (Goldberg, 1989), where P_t represents the population of chromosomes at generation t. The number of chromosomes in the population of the GA remains fixed from generation to generation.

```
procedure genetic_algorithm
    begin
        t:=0;
        initialise P_t;
        evaluate P_t;
        while (not stopping-condition) do
        begin
            select P_t+1 from P_t;
            t:=t+1;
            crossover P_t;
            mutate P_t;
            evaluate P_t;
        end
    end
```

Figure 2.1 - The Structure Of The Simple Genetic Algorithm

The steps within the GA are explained in greater detail:-

Initialisation - The first step of the GA is to generate the initial population of chromosomes. In general, this involves choosing a random allele for each gene of each chromosome. This is repeated for successive chromosomes until a population of individuals is produced. The size of the population can determine the quality of convergence and is problem dependant (Goldberg, et. al. 1992).

Evaluation - The evaluation phase of the GA determines the relative fitness of the chromosomes within the population (Goldberg & Rudnick, 1991 (c)). In general, this is equivalent to the object value of the parameter set represented by that chromosome. The relative fitness of a chromosome determines its survival and possible propagation in subsequent generations. Once calculated, the fitness is stored alongside the chromosomes for use by the selection algorithm.

Selection - The selection algorithm determines which of the chromosomes of the current population are represented in the following population. Typically, the selection process will ensure that those chromosomes of high fitness prosper at the expense of those

chromosomes of low fitness. The most commonly used technique is the Roulette Wheel Selection algorithm (Goldberg, 1989) which, for each free space in the new population, statistically selects a chromosome from the current population according to its relative fitness. This process may be compared with spinning a weighted roulette wheel for each member of the new population. The proportion of the wheel assigned to each chromosome in the current population is determined by that chromosomes contribution to the total fitness of the population. Other techniques include Tournament selection (Goldberg, et. al. 1991(c)) which generates the new population by choosing, for each free space in the new population, the fittest of a randomly selected subset of the current population. In general, two competitors are used giving a binary tournament. Linear normalisation (Goldberg, 1989) ranks and assigns a fitness value according to the relative position of the individual within the population, it sorts them in ascending or descending order. Linear fitness scaling and power fitness scaling (Davis, 1991) use sorting and then either scale the fitness using a linear or a power law to modify the sorted fitness value. Stochastic remainder selection (Davis, 1991) is a variant of the Roulette Wheel Selection algorithm, which guarantees that a chromosome will receive at least the integer part of its expected number of offspring.

Crossover - The crossover operator, generally held to be the principal genetic operator of the GA (Schaffer & Eshelman, 1991), combines the genetic information of a pair of the parent individuals (or chromosomes) to produce a pair of offspring chromosomes. These offspring then take the place of the parent chromosomes in the current population. This exchange of genetic information is achieved by various methods. Single Point Crossover randomly chooses a locus and swaps between the parent chromosomes the 'bits' or allele values of each following gene. Two Point Crossover uses two crossover points and all genes between the crossover points exchange allele values. Uniform Crossover extends this notion further allowing each gene to retain or swap allele values with equal probability.

Other crossover operators include Average Crossover and Arithmetic Crossover. The proportion of the population selected for crossover is known as the crossover rate and is generally set at about 60% (Goldberg, 1989). The parent chromosomes are usually selected in advance so that no individual chromosome takes part in more than one crossover event.

Other crossover operators have been suggested which include a degree of mutation (Jones 1995) who demonstrated that a macromutation with the mechanical form of crossover that substitutes a randomly constructed parent as one of the recombinants, performs as well as and occasionally better than crossover when clearly defined building blocks are not present. This macromutation named *headless chicken crossover*, had the mechanical form of crossover, i.e. the transfer of genetic material between two parents, but paired each population member with a randomly generated parent rather than a parent chosen from the population. Jones shows that on problems where well-defined building blocks do not exist the macromutation performs better than the GA with crossover. The effectiveness of this operator calls into question the range of problems for which crossover is well suited and suggests that macromutations are often sufficient to solve difficult problems.

Mutation - Unlike crossover, the mutation operator acts upon single chromosomes chosen at random from the population. For each selected chromosome a random locus is selected, and the allele value of the gene at that locus is altered. This new chromosome replaces its parent in the population. The proportion of the total number of genes in the population selected for mutation is known as the mutation rate and is generally inversely proportional to the population size (Goldberg, 1989). Unlike crossover, it is not usual to pick the target genes in advance, and it is therefore possible that the same gene may be subject to more than one mutation event.

The Stopping Condition - A number of criteria may be used to halt the GA search process. For example,

- The GA executes for a pre-set number of generations.
- The maximum or average fitness of the population reaches a pre-set target.
- The population converges (all chromosomes within the population are identical).

GA's vary from practitioner to practitioner, and the GA outlined above can be considered as a *simple genetic algorithm.* There are various other techniques that have been used to enhance the performance of the GA and these are briefly discussed.

Elitism

The best member of a population may fail to produce offspring in the next generation. The *elitist* strategy fixes this potential source of loss by copying the best member of each generation into the succeeding generation. A disadvantage of elitism is that it can increase the probability of domination of a population by a super individual, however, used discriminately it does improve GA performance.

Steady-State Reproduction

When a GA reproduces, it replaces a predefined percentage of parent individuals by their children. This generational replacement technique has some potential drawbacks. One is that even with an elitist strategy, many of the best individuals found may not reproduce at all, and their genes may be lost. It is also possible that mutation or crossover may alter the best chromosomes genes so that good features are destroyed. Neither of these outcomes is desirable. One solution to this problem is to modify the reproduction technique so that only one or two individuals are replaced at a time. This is termed steady-state reproduction (Syswerda, 1989).

Representation Issues

Bit string encoding is the most common encoding technique used by genetic algorithm researchers. Bit strings have several advantages over other encodings, they are simple to create and manipulate, and they are theoretically tractable, which leads to schemata theory.

Schemata theory was first described by Holland (Holland, 1975), schemata are similarity templates for strings. Each schema defines a subset of strings with identical values at specified string locations, and provides a means by which similarities among the individual population members can be described and exploited.

In order to define schemata, the 'alphabet' of the strings is used to define values at specific locations, and an additional character, the symbol (#), is used as a 'wild card' in locations where the value does not matter. Schemata can thus generally be thought of as comprising an alphabet of $a_0 + 1$ characters, where a_0 is the number of characters in the GA representation. The GA strings are usually represented in binary, so the schemata comprises the characters {0,1, #}.

As an example, consider the schemata of length 4 that may appear in, say, the leftmost four positions of an individual within a population. One such schema is #000, which has two member strings. That is, two strings match the schema, 0000 and 1000. The schema 1##0 has four matching strings, 1000, 1010, 1100, and 1110. For a string of length l and an alphabet of a_0, there are $(a_0 + 1)^l$ total possible schemata. Another useful measure is the total possible number of unique schemata in a population. Consider a specific string of length 8, since each string position can assume the value it has, or the wild card value, the string belongs to $2^8 = 256$ schemata. Any binary string of length l thus belongs to 2^l schemata. Populations with a high diversity have more schemata.

At this point it is useful to reconsider the basic GA operations of reproduction, crossover and mutation. Schemata that are part of an individual with high fitness will be reproduced more often than average, therefore highly fit schemata benefit from reproduction. If reproduction were the only operator used, though, no new regions of the search hyperspace would ever be explored. Crossover and mutation guide the search into new regions.

Crossover, however, is a slightly more complicated matter than reproduction. Consider two schemata, 1######0 and ###10###. If both are part of strings of equal fitness, one point or two point crossover is more likely to disrupt the first, since it is likely that a crossover point will occur between the two string endpoints. The second is more compact and is relatively unlikely to be disrupted by a one or two point crossover operation.

Mutation is not likely to disrupt either schema, since it typically occurs at a very low rate, and since it is considered on a bit by bit basis, it is just as likely to disrupt one as the other.

While crossover and mutation are potentially disruptive, they facilitate an efficient search. Furthermore, compact (short) schemata that are part of highly fit individuals will, with high probability, appear in ever-increasing numbers in future generations. The schemata are the elements of which future generations are built, Holland (Holland, 1962) named them 'building blocks.' The schema theorem provides a quantitative estimation of one aspect of GA performance.

The Schema Theorem

The schema theorem predicts the number of times a specific schema will appear in the next generation of a GA, given the fitness of the population members containing the schema, the average fitness of the population, and other parameters. The GA is effectively working with

a large number of schemata simultaneously, ranging from very short schemata to schemata as long as the individual population member. The schema theorem provides a quantitative prediction for all schemata, regardless of length. The schema theorem (Goldberg, 1989) is:-

$$n_{t+1}(S) \geq n_t(S)\frac{f(S)}{f(avg)}\left[1 - p_c\frac{\delta(S)}{l-1}\right] - o(S)p_m \qquad (2.4)$$

Where, n is the total number of examples of a particular schema S. The subscript $t+1$ and t refer to time steps, or generations. The parameter $f(S)$ is the average fitness of the individual, while f_{avg} is the average fitness of the entire population. The probabilities of crossover and mutation are p_c and p_m respectively.

The parameter $\delta(S)$ is called the defining length of the schema, it is the distance between the first and last specific string positions. For example, for the schema #01#11#, the defining length is 4. The total length of the string is l, while $o(S)$ is the *order* of the schema, or the number of fixed positions (0's and 1's) in the schema. In the preceding example, the order of the schema is 4. The order of a schema is the number of potential 'cut' points within the schema that could be affected by crossover.

Schemata are used to attempt to explain why GAs work, it is based on the idea that GAs solve problems by hierarchically composing relatively fit, short schemata to form complete solutions (Building Block Hypothesis). This theory is an approximation and it is not generally accepted as positive proof of how a GA works.

Another representation method is the use of real number encoding techniques. This replaces bit strings with real numbers and consequently requires a revised mutation and crossover operator to manipulate them (Davis, 1991). The main advantages of using real number

encoding are that numerical representation is effective on mathematical problems, mathematical operators, and numerical operators may greatly improve the performance of the GA on numerical problems.

Genetic algorithms directly manipulate a coded representation of the problem. The GA operating on fixed-length representations is capable of solving many problems. For many cases, this is not the most natural representation for a solution. The size and shape of the solution is not known in advance, so the program should have the potential of changing its size and shape. One approach to this problem is to use a Structured Genetic Algorithm (StGA) (Dasgupta, 1992) where discrete features of the problem are encoded in blocks which are either turned on or off depending on the initial block coding. The active parameters are passed to the evaluation model and this provides a variable length representation using GAs.

2.6.2 Neural Networks

Neural networks (NNs) excel at recognition and classification types of problems, and can be applied to the systems identification problem using adaptive algorithms for either parameter or functional estimation (Tenorio & Lee, 1990). Neural networks (NNs) are information processing systems. In general, neural networks can be thought of as "black box" devices that accept inputs and produce outputs. In the simplest terms, neural networks map input vectors onto output vectors. Some of the operations that neural networks perform include:

- *Classification* - An input pattern is passed to the network, and the network produces a representative class as output.

- *Pattern matching* - An input pattern is passed to the network, and the network produces the corresponding output pattern.

- *Pattern completion* - An incomplete pattern is passed to the network, and the network produces an output pattern that has the missing pattern portions filled in.

- *Noise removal* - A noise-corrupted input pattern is presented to the network, and the network removes some (or all) of the noise and produces a cleaner version of the input pattern as output.

- *Optimisation* - An input pattern representing the initial values for a specific optimisation problem is presented to the network, and the network produces a set of variables that represent an acceptably optimised solution to the problem.

- *Control* - An input pattern is presented that represents the current state of a controller and the desired response for the controller, and the network output is the proper command sequence that will create the desired response.

- *Simulation* - An input pattern is presented that represents the current state vector of a system or time series. The trained network generates structured sequences or patterns that simulate behaviour of the system.

Neural networks consist of processing elements and weighted connections. Figure 2.2 illustrates a typical neural network.

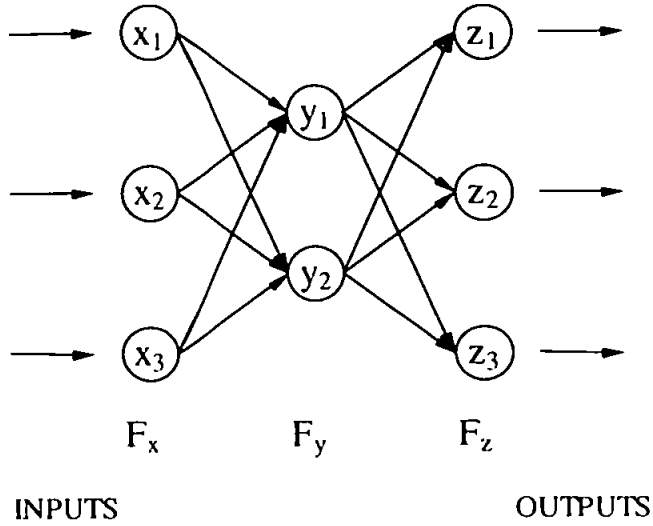


Figure 2.2 - A Typical Neural Network

Each layer in a neural network consists of a collection of processing elements (PEs). Each PE collects the values from all of its input connections, performs a predefined mathematical operation (such as a dot-product followed by a threshold), and produces a single output value (Pandya & Macy, 1995). The neural network in figure 2.2 has three layers: F_x, which consists of the PEs $\{x_1,x_2,x_3\}$; F_y, which is called a hidden layer and consists of the PEs (y_1, y_2) and F_z, which consists of the PEs $\{z_1,z_2,z_3\}$ (from left to right, respectively).

PEs are joined with weighted connections. In figure 2.2 there is a weighted connection from every F_x PE to every F_y PE, and there is a weighted connection from every F_y PE to every F_z PE. Each weighted connection (often referred to as either a connection or a weight) acts as both a label and a value. As an example, in figure 2.2 the connection from the F_x PE *x*1 to the F_y PE *y*2 is the connection weight W21 (the connection from *x*1 to *y*2). In most uses of NNs connection weights store the information, or knowledge, in a network. The values of the connection weights are often determined by a neural network learning procedure. It is through the adjustment of the connecting weights that the neural network is able to learn. By performing the update operations for each of the PE's when an input pattern is presented, the neural network is able to recall information.

There are several important features illustrated by the neural network shown in figure 2.2 that apply to all neural networks (Welstead, 1994):

- Each PE acts independently of all others - each PE's output relies only on its constantly available inputs from the abutting connections.
- Each PE relies only on local information - the information that is provided by the adjoining connections is all a PE needs to process: it does not need to know the state of any of the other PE's to which it does not have an explicit connection.
- The large number of connections provides redundancy and facilitates a distributed representation.

The first two features allow neural networks to operate efficiently in parallel. The last feature provides neural networks with inherent fault-tolerance and generalisation qualities that are very difficult to attain with most other computing systems. In addition to those features, by properly arranging the topology of the networks, introducing a nonlinearity in the processing elements (i.e., adding a nonlinear threshold function), and by using appropriate learning rules, neural networks are able to learn arbitrary nonlinear mappings. This is a powerful attribute. There are three situations where neural networks are advantageous: -

1. Situations where relatively few decisions are required from a massive amount of data (e.g. speech and image processing);
2. Situations where nonlinear mappings must be automatically acquired (e.g. loan evaluations and robotic control); and

3. Situations where a near-optimal solution to a combinatorial optimisation problem is required very quickly (e.g., job shop scheduling and telecommunication message routing).

Neural networks comprise of three principal elements needed to specify the network:

- *Topology* - how a neural network is organised into layers and how those layers are connected.

- *Learning* - how a neural network is configured to store information.

- *Recall* - how the stored information is retrieved from the network.

Neural networks (Pandya & Macy, 1995) can also be used for curve fitting, surface fitting and other regression problems. Design of a neural network for pattern classification may be viewed as a curve-fitting problem in hyperspace, where learning weights amounts to finding a hyper-surface that provides a 'best fit' to a given set of training data.

Radial basis networks

Radial-basis functions (RBF) (Pandya & Macy, 1995, Spect, 1990) provide a technique for interpolation in a high-dimensional space. RBF's construct local approximations using exponentially decaying localised nonlinearities based on a Gaussian function in two dimensions.

RBF networks have a static Gaussian function as the nonlinearity for the hidden layer processing elements. The Gaussian function responds only to a small region of the input space where the Gaussian is centred. The key to a successful implementation of these networks is to find suitable centres for the Gaussian functions. This can be done with supervised learning, but an unsupervised approach usually produces better results.

The simulation starts with the training of an unsupervised layer. Its function is to derive the Gaussian centres and the widths from the input data. These centres are encoded within the weights of the unsupervised layer using competitive learning. During the unsupervised learning, the widths of the Gaussians are computed based on the centres of their neighbours. The output of this layer is derived from the input data weighted by a Gaussian mixture. Once the unsupervised layer has completed its training, the supervised segment then sets the centres of Gaussian functions (based on the weights of the unsupervised layer) and determines the width (standard deviation) of each Gaussian. Any supervised topology such as a multilayer perceptron, (MLP), may be used for the classification of the weighted input. A typical RBF network topology is shown in figure 2.3.

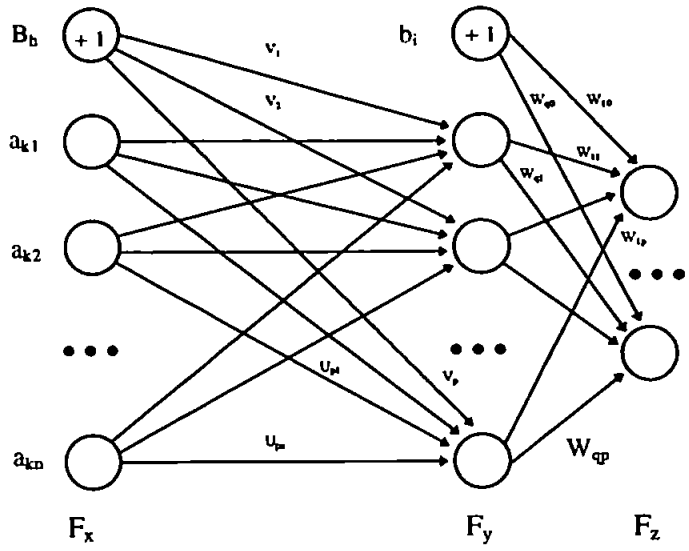


Figure 2.3 - Radial Basis Function Network Topology

The advantage of the radial basis function network is that it finds the input to output map using local approximators. Usually the supervised segment is simply a linear combination of the approximators. Since linear combiners have few weights, these networks train extremely fast and require fewer training samples (Eberhart, Simpson & Dobbins, 1996).

Some of the advantages and disadvantages of radial basis function networks are listed below (Welstead, 1994).

Advantages include: -

- The ability to create nonlinear decision boundaries
- Verification and validation are possible
- Networks output provides graded membership information and novelty detection
- Does not experience local minima problems of back-propogation
- The LVQ learning phase is relatively insensitive to the order of pattern presentation

Disadvantages include: -

- The training process can be somewhat slow
- Several parameters require "tuning"
- It is difficult to perform incremental learning
- It is difficult to process missing and weighted features

Radial basis function networks are being used for an increasing number of applications. They are computationally easy to train, and performance often equals or exceeds that of other paradigms (Welstead, 1994).

2.7 Systems Identification Using Genetic Algorithms and Neural Networks

Genetic algorithms can be used for systems identification but the main drawback is the inflexibility of the structure of the answer. To use a GA for curve fitting we could assume that the answer is a 5^{th} order polynomial and code the GA to represent the coefficients for the polynomial. This will produce a solution to the problem, but by stating the structural form of the final equation, the answer is limited to the initial function chosen. The use of a structured GA does allow some flexibility of the structure of the equations, but to allow enough variation of equations in the encoding there would be a large amount of redundant information within the answer which would effect the efficiency of the algorithm. NNs are very good at finding relationships between sets of data. The major drawback with this technique is that there is no known way to represent the results of a run as a mathematical equation.

2.8 Summary

A method is required which will not only evolve the variables within the function but also the functional form of the equations. Standard mathematical techniques require the user to assume the functional form of the solution before any analysis can start and the spline approach will not produce a continuous function which describes the data. It is possible to use a structured genetic algorithm to perform symbolic regression but it would involve large structures in order to represent the equations, resulting in large amounts of redundancy within the structures undergoing adaptation. As stated earlier, the size and shape of the structures should not be specified in advance, they should emerge during the problem-solving process as a result of the demands of the problem. Neural networks provide a mathematically proved method for solving *any* problem, the major drawback being that the results of the network are virtually impossible to view and represent as a mathematical function. Genetic programming (GP) can provide interpretable equations and does not require any prior knowledge of the system as in the case of a GA. The structures that are produced can dynamically vary in size and shape and so the GP paradigm will be used for the solution to symbolic regression problems.

CHAPTER 3

GENETIC PROGRAMMING

This chapter introduces the Genetic Programming paradigm in detail and includes representation issues, genetic operators and theoretical research attempting to explain the mechanisms of the method. The recently developed GP paradigm (Koza, 1992 and Koza, 1994) is a method of program induction, which genetically breeds a population of computer programs to solve problems.

3.1 The Genetic Programming Paradigm

The GP paradigm deals with the problem of representation in GA's by increasing the complexity of the structures undergoing adaptation. In particular, the structures in GP are general, hierarchical computer programs of dynamically varying size and shape.

GP commences with an initial population of randomly generated computer programs composed of functions and terminals appropriate to the problem domain.

3.2 Outline Of The Standard GP Algorithm

The GP algorithm is similar to the GA algorithm, the only difference being in the implementation of various aspects of the algorithm such as crossover and mutation, due to the structures used to represent the solutions to a given problem. Figure 3.1 shows the structure of the Genetic Programming paradigm, where P_t represents the population of chromosomes at generation t. The number of chromosomes in the population of the GP remains fixed from generation to generation. The first step is the initialisation of the population followed by the evaluation where the population is ranked in order of a specified

fitness measure. The next generation is then selected from the current generation and crossover and mutation operators are applied to the population. The process then repeats until a prespecified stopping criteria has been met.

```
procedure genetic_programming
    begin
        t:=0;
        initialise P_t;
        evaluate P_t;
        while (not stopping-condition) do
        begin
            select P_t+1 from P_t;
            t:=t+1;
            crossover P_t;
            mutate P_t;
            evaluate P_t;
        end
    end
```

Figure 3.1 - The GP Algorithm

3.3 The Structures Undergoing Adaptation

In every adaptive system or learning system, at least one structure is undergoing adaptation. For the conventional genetic algorithm and genetic programming, the structures undergoing adaptation are a *population* of individual points from the search space, rather than a single point. Genetic methods differ from most other search techniques in that they simultaneously involve a parallel search involving many points in the search space. The functions used may be standard arithmetic operations, programming operations, mathematical functions, logical functions, or domain-specific functions. Depending on the particular problem, the computer program may be Boolean, integer, real, complex, vector, symbolic, or multiple valued. The creation of the initial random population is a blind random search of the problem search space.

The set of possible structures in genetic programming is the set of all possible compositions of functions that can be composed recursively from the set of N_{func} functions from:

$$F = \{f_1, f_2, \ldots, f_{N_{func}}\} \tag{3.1}$$

and the set of N_{term} terminals from:-

$$T = \{a_1, a_2, \ldots, a_{N_{term}}\} . \tag{3.2}$$

Each particular function f_i in the function set F takes a specified number $z(f_i)$ of arguments $z(f_1), z(f_2), \ldots, z(f_{N_{func}})$ That is, function $f_i$has arity $z(f_i)$. The arity being the number of arguments taken by the function.

The functions in the function set may include:-

- arithmetic operations (+ , - , * ,etc.),
- mathematical functions (sin, cos, exp, log),
- Boolean operations (AND, OR, NOT),
- conditionals operators (If-Then-Else),
- functions causing iteration (Do-Until),
- functions causing recursion, and
- any other domain-specific functions that are defined.

The terminal set T is typically composed of either variable atoms (representing, perhaps, the inputs, sensors, detectors, or state variables of some system) or constant atoms (such as the number 3.0 or the Boolean constant NIL).

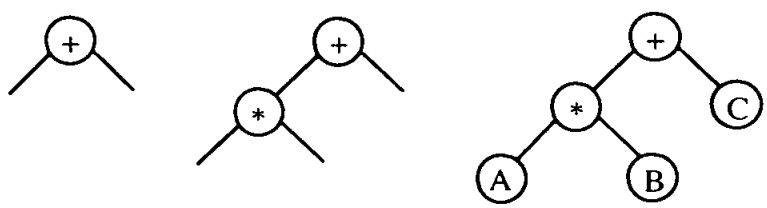
3.4 Closure Of The Functional Set And Terminal Set

The *closure* property requires that each of the functions in the function set is able to accept, as its arguments, any value and data type that may possibly be returned by any function in

the function set and any value and data type that may possibly be assumed by any terminal in the terminal set. That is, each function in the function set should be well defined and closed for any combination of arguments that it may encounter.

3.5 Initial Structures

The generation of each individual in the initial population is achieved by randomly generating a rooted, point-labelled tree with ordered branches. The process begins by selecting one of the functions from the set F at random to be the label for the root of the tree. The selection of the label is restricted to the set of functions because hierarchical structures are required, not a degenerate structure consisting of a single terminal. Figure 3.2(a) shows the beginning of the creation of a random program tree. The function + (arity 2) was selected from a function set F as the label for the root of the tree.



3.2(a) **3.2(b)** **3.2(c)**

Figure 3.2 - Initial Structure Formation

Whenever a point of the tree is labelled with a function f from F, then $z(f)$ lines, where $z(f)$ is the number of arguments taken by the function f, are created to radiate out from that point. Then, for each such radiating line, an element from the combined set $C = F \cup T$ of functions and terminals is randomly selected to be the label for the endpoint of that radiating line. If a function is chosen to be the label for any such endpoint the generating process continues recursively as described above. For example figure 3.2(b) shows the function * (multiplication, arity 2) from the combined set $C = F \cup T$ of functionals and terminals

selected as a label of the internal nonroot point at the end of the first line radiating from the function +. Since a function was selected, it will be an internal, non-root point of the tree that will eventually be created. The function * takes two arguments, therefore figure 3.2(b) shows two lines radiating out from point 2. If a terminal is chosen to be the label for any point, that point becomes an endpoint of the tree and the generating process is terminated for that point. For example figure 3.2(c) shows a terminal A from the terminal set *T* selected to be the label of the first line radiating from the point labelled with the function. This process continues recursively from left to right until a completely labelled tree has been created. *. In figure 3.2(c) the terminals B and C are selected to be the labels of the two other radiating lines.

This generative process can be implemented in several different ways resulting in initial random trees of different sizes and shapes. Two of the basic ways are called the 'full' method and the 'grow' method (Koza, 1992). The depth of a tree is defined as the length of the longest non-backtracking path from the root to an endpoint. The 'full' method of generating the initial random population involves creating trees for which the length of every non-backtracking path between an endpoint and the root is equal to the specified maximum depth. This is accomplished by restricting the selection of the label for points at depths less than the maximum to the function set *F*, and then restricting the selection of the label for points at the maximum depth to the terminal set *T*. A tree with a maximum depth of 2 will have 1 element at layer 1, and 2 elements at layer 2, giving a tree of length 3. A tree of maximum depth 3 will have 7 elements, and a tree of maximum depth n will have $(2^n - 1)$ elements.

The 'grow' method of generating the initial random population involves generating trees that are variably shaped. The length of a path between an endpoint and the root is no

greater than the specified maximum depth. This is accomplished by making the random selection of the label for points at depths less than the maximum from the combined set $C = F \cup T$ consisting of the union of the function set F and the terminal set T, while restricting the random selection of the label for points at the maximum depth to the terminal set T. The 'ramped half-and-half' generative method (Koza, 1992) is used on all problems within GP. This is a mix of the 'full' and 'grow' methods creating trees having a wide variety of sizes and shapes. When generating the initial population a proportion are generated using the 'full' method and the rest by the 'grow' method, the proportion of each is usually set at 50% (Koza, 1992).

3.6 Primary Operations For Modifying Structures

Two primary operators are used to modify the structures undergoing adaptation in GP, and are discussed in the next sections. The two main opetators are:

- Darwinian reproduction
- Crossover (sexual recombination).

3.6.1 Reproduction

The reproduction operators that can be used are the same as those used for GA's, these include:

- Fitness-proportionate reproduction.
- Rank selection.
- Tournament selection.

3.6.2 Crossover

The crossover (recombination) operation for GP creates variation in the population by producing new offspring that consist of parts taken from each parent (Spears & Anand. 1991). The crossover operation starts with two parental expressions and produces two offspring expressions. The first parent is chosen from the population by the same fitness-based selection method used for the reproduction operator, as is the second parent.

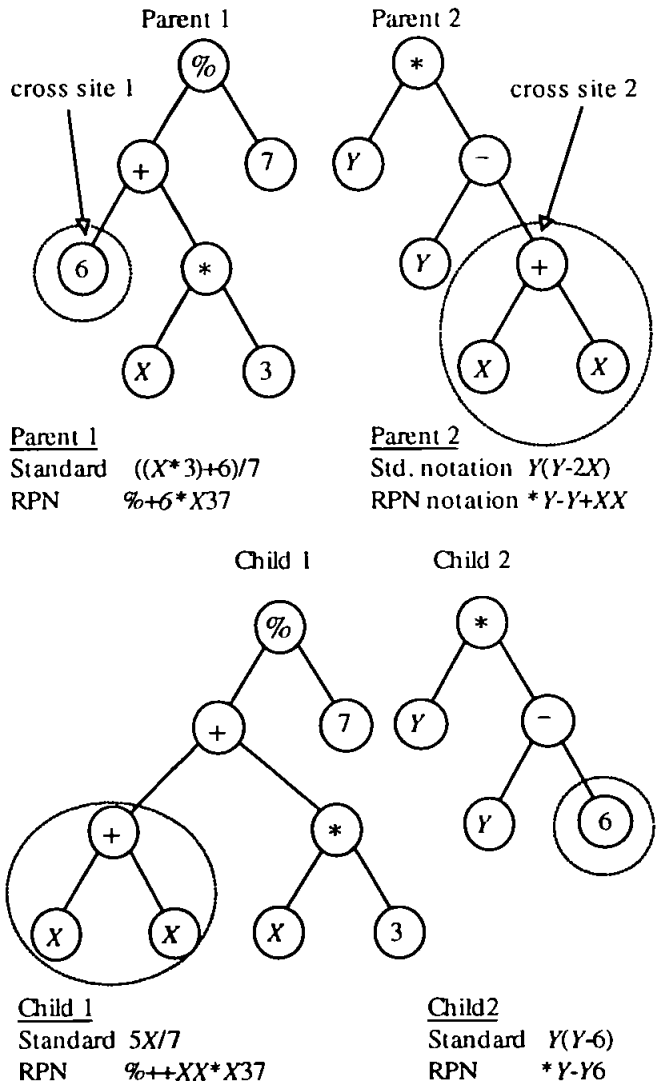


Figure 3.3 - The GP Crossover Operator

The operation begins by independently selecting, using a uniform probability distribution, one random point in each parent to be the crossover point for that parent. Note that the two

parents typically are of unequal size. The first offspring expression is produced by deleting the crossover fragment of the first parent from the first parent and then inserting the crossover fragment of the second parent at the crossover point of the first parent. The second offspring is produced in a symmetric manner. For example, consider the two parent symbolic-expressions shown in figure 3.3. Parent 1 has a terminal, the real number 6, located at the first crossover point and parent 2 has a sub tree located at its crossover point which represents ($2X$). The two are exchanged to produce two new individuals, Child 1 and Child 2. If terminals are located at both crossover points, the crossover operator just swaps these terminals from tree to tree. The effect of crossover, in this event, is akin to a point mutation. Thus, occasional point mutation is an inherent part of the crossover operator. Other types of crossover include context preserved crossover (D'haeseleer, 1994) which attempts to preserve the context in which subtree appeared in the parent trees.

Recently, a similar conclusion to that of GA crossover (Jones, 1995) has been reached for genetic programming using subtree crossover (Angeline, 1997). Angeline demonstrated that two types of headless chicken crossover defined for subtrees performed equivalently to standard subtree crossover when compared using three different problems.

3.7 Secondary Operators

In addition to the two primary genetic operators of reproduction and crossover in GP, there are optional secondary operators that can also be used in the optimisation process. The most important of these is mutation.

3.7.1 Mutation

When using Gas employing a binary representation, mutation is referred to as 'bit-flipping', but in GP a mutation is the manipulation of a structure and has been described as a random substitution of a sub-tree with another sub-tree. Branch mutation can be implemented where a complete sub-tree is replaced with another (similar to the crossover operator). Alternatively node-mutation can be introduced which applies a 'random' change to a single node, replacing its original by another value. Branch mutation is essentially a form of crossover and as such is not used, node-mutation is used but when implementing node-mutation it is very important to only mutate terminals with other terminals of the same arity (number of branches) and functionals into other functionals of the same arity. The two cannot be mixed as closure will not be achieved and the structures will not be well defined.

Other mutation operators have been used within GP (Chellapilla, 1997) which uses 6 tree mutation operators with no crossover. Chellapilla's results indicate that the mutation operators produced results comparable to those of Koza, (Koza, 1992) and in many cases offered improved cumulative probabilities of success and fewer required evaluations to produce an individual of the same quality.

3.8 Computer Representation Of Structures

The Symbolic-expressions (S-expressions) representing evolved functions are coded from the tree structures into Reverse Polish Notation (RPN). This dispenses with the need for brackets and there is a one-to-one relationship between RPN and standard notation

All work produced by Koza (Koza, 1992, and Koza, 1994) uses the LISP language. The language used for all runs presented here is C++, and as a result a method of structure representation is required. Each individual S-expression is stored in an array and a

maximum limit is set for the length (typically set to 100 elements). The elements within the individual were initially represented by a non-signed integer giving 65,536 possible values. These are then segmented to represent the sets of terminals and functions. For example the integers 1-10 are allocated to functions, 11-20 for terminals and 21-65536 for real numbers. This would allow 65515 possible values for representing real number, and although this seems adequate, it does present problems. Suppose a GP run is required to optimise a set of data. The *expected* range of the real numbers would have to be predefined, say -10.0 to 10.0, this would give a range of 20.0 and so the maximum resolution that can be represented is ± 0.00030527 (given by 20.0/65515).

For most applications this level of accuracy will be sufficient, but what if the initial range of values expected is incorrect. If the solution requires a real number greater than 10.0, the only way to represent this will be by using a sub tree representing the addition of two real numbers, perhaps (9.4 + 5.7), which will add complexity to the system under investigation. The problem is overcome by using an array of unsigned characters to represent elements within the individual S-expression, and another array of the same size for floating point numbers to represent the set of real numbers. This gives a maximum of 256 values for representing both functionals and terminals. The functionals are allocated values from 1 - 99, and terminals 100-255 (the value 0 is used to represent empty spaces at the end of the S-expression). The real numbers have to be within the range allocated for terminals (100-255) and a value of 100 is used to represent *all* real numbers. The actual real number value is stored in the array of floating point number. As an example suppose the following tree structure shown in figure 3.4 is produced. Written in standard notation the equation would be:- $Y (X + 3.7)$, but in RPN the structure is:- * + X 3.7 Y, this has the immediate advantage that parenthesis are no longer necessary, and there is a one-to-one correspondence between the standard notation and RPN of algebraic formula .

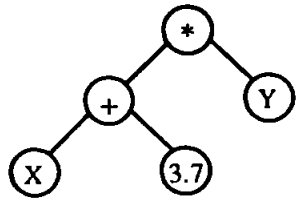


Figure 3.4 - Example of Symbolic-expression Tree

Figure 3.5 shows an individual together with its coded values. When a value of 100 is received from the character array (a real numbered terminal) the location of the real number is stored at the same point as the character of value 100 but in the floating number array. The maximum length of any given individual is limited and is typically set at 100 elements.

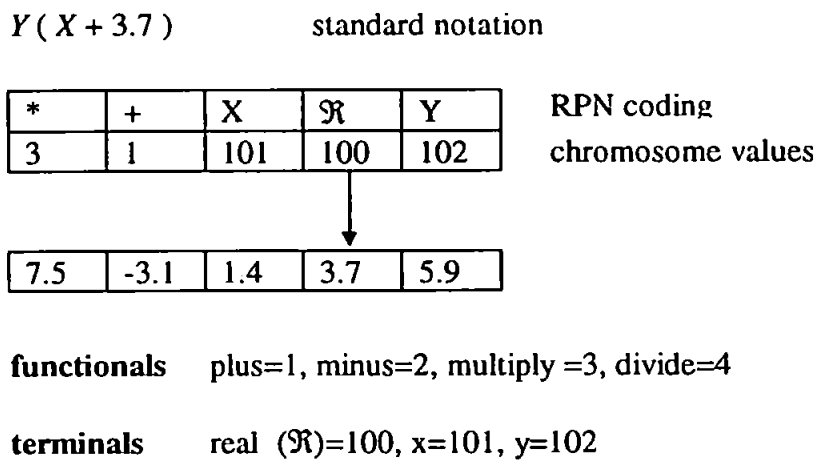


Figure 3.5 - Representation of Structures

3.9 GP Schemata Theory

The first attempt to produce a schema theory for GP was made by Koza (Koza, 1992), who produced an informal argument showing that Holland's schema theorem would apply to GP as well. The argument was based on the idea of defining a schema as the subspace of all trees which contain, as subtrees, a predefined set of complete subtrees. According to Koza's definition, a schema H is represented as a set of symbolic expressions, e.g. H = { (+ 1 x), (* x y) } represents all the programs including at least one occurrence of (+ 1 x) and one of (* x y).

Koza's work was later formalised and refined into a schema theorem for GP (O'Reilly & Oppacher, 1995). A schema was defined as an unordered collection (a multiset) of subtrees and tree fragments. Tree fragments are trees with at least one leaf that is a 'don't care' symbol ("#") which can be matched by any subtree. For example the schema H = { (+ # x), (* x y), (* x y) } represents all the programs including at least one occurrence of the tree fragment (+ # x) and at least *two* occurrences of (* x y).

This definition of schema allowed the introduction of the concept of order and defining length for GP schemata. The first real attempt at producing a viable schema theory of GP was produced (Poli & Langdon, 1997) with the theory being based on a new simpler definition of the concept of schema for GP which is very close to the original concept of schema in GA's. The theory is based around one-point crossover and point mutation, and results published show that the conjectures are correct.

3.10 Summary Of The GP Paradigm

This chapter outlines the processes involved in the GP paradigm. A wide variety of different problems from different fields have been solved (Koza, 1992, 1994) and provides considerable evidence for the generality of the genetic programming paradigm. The fact that the output of genetic programming is always a computer program in the form of its own parse tree means that the result can be immediately executed as a computer program, and although the output can be complex, it is generally easy to apply straightforward simplification and optimisation. Genetic programming also requires little prior knowledge of the problem, unlike other evolutionary computing techniques. Neural networks requires numbers of layers, processing units at each layer and connectivity, and genetic algorithms require predefined structures and can only provide limited variation of string sizes. The information that is required by genetic programming such as the choice of terminal set and the set of primitive functions is also required by every other paradigm for machine learning. In conclusion, genetic programming is a robust and efficient paradigm for discovering computer programs using the expressiveness of symbolic representation. The technique has solved various problems including sequence induction, planning, symbolic regression, automatic programming, and evolution of emergent behaviour (Koza, 1992, 1994).

CHAPTER 4

COMPARISON OF TECHNIQUES

The GP paradigm along with other techniques will now be used on various test functions, starting with curve fitting, to assess the viability of these methods for symbolic regression purposes.

4.1 Curve Fitting

The first example uses one independent, and one dependant variable. This is the simplest form of symbolic regression, the function to be discovered being of the form: -

$$y = f(x). \tag{4.1}$$

In order to illustrate applications of the various techniques to curve fitting a quartic test function is considered: -

$$y = ax^4 + bx^3 + cx^2 + dx + e \tag{4.2}$$

Where: - $x \in [-5.0,5.0]$, and, a = 0.030, b = 0.050, c = - 0.700, d = 0.100 and e = 8.600.

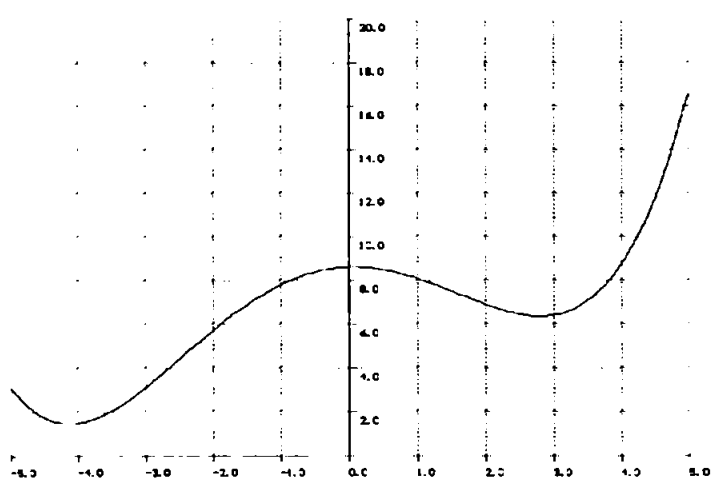


Figure 4.1 - Quartic Test Equation

The test function is shown in figure 4.1. The test data consists of 11 points equally spaced between the minimum and maximum x-axis values. Genetic programming will be the first method used to solve this problem and the process can be broken down into three steps.

The first step in using genetic programming is to identify the set of terminals, and the information, which the mathematical expression must process, is the value of the independent variable x. The test function also includes real numbers and so the set of real numbers is included. Thus, the terminal set is

$$T = \{ x, \Re \}. \tag{4.3}$$

Where $\Re$ is the set of real numbers.

During the initialisation of the population if a real number is chosen as a terminal then a number is randomly chosen between a predefined range, in this case real numbers are within the range -10.0 to 10.0.

The second major step in preparing to use genetic programming is to identify the set of functions that are used to generate the mathematical expressions that attempt to fit the given finite sample of data. If knowledge that the answer is $ax^4+bx^3+cx^2+dx+e$ is used, a function set consisting of only addition and multiplication operations would be sufficient for this problem. A more general choice might be the functional set consisting of the four ordinary arithmetic operators of addition, subtraction, multiplication, and the protected division function % (in this context, protected means the function is protected from division by zero). Initial testing of the technique using this functional set, i.e.

$$F = \{ +, -, *, \% \} \tag{4.4}$$

Produced solutions which were no more than a linear fit to the curve, if a wider variety of problems is to be solved, the functional set could also include the sine function SIN, the cosine function COS, the exponential function EXP, and the protected logarithm function RLOG (Koza, 1992). The functional set for this problem is thus:

$$F=\{ +, -, *, \%, \text{SIN}, \text{COS}, \text{EXP}, \text{RLOG}\} \tag{4.5}$$

Taking two, two, two, two, one, one, one, one arguments respectively.

The third major step in preparing to use genetic programming is to identify the fitness measure. The raw fitness for this problem is the root mean squared (RMS) of the difference (error) between the value in the real-valued range space produced by the expression for a given value of the independent variable x_i and the correct y_i in the range space. The closer this sum is to zero, the better the computer program. Error-based fitness is the most common measure of fitness used in this thesis. The RMS fitness is given by: -

$$f = \sqrt{\frac{\sum_{i=n}^{i=0} (x_i - x_i^*)^2}{n}} \tag{4.6}$$

where:-

f = fitness $\quad n$ = number of samples $\quad x_i$ = evolved solution $\quad x_i^*$ = exact solution

Note:-As in the GA , this fitness measure is used throughout unless stated otherwise, therefore establishing a minimisation problem.

A population size of 500 is used with a crossover rate of 0.6, a mutation rate of 0.002 and an initial population generated by using the 'ramped half-and-half' method described in section 3.5. The maximum number of generations is set at 100 and the maximum chromosome length is set to 100 elements. The selection method used is roulette wheel selection and the best individual is always preserved (elitism=1). The results of 10 runs of genetic programming are shown in table 4.1.

Run no.	RMS error	Individual length
1	0.137129	99
2	0.0852	98
3	0.06549	88
4	0.04432	99
5	0.07655	86
6	0.01834	98
7	**0.005441**	**96**
8	0.211774	92
9	0.490605	97
10	0.09462	95
Avg.	0.122947	94.8

Table 4.1 Results Of GP On Quartic Test Function

Although GP evolves solutions to this problem which have reasonable fitness, two important points are illustrated. The first is that GP has to be run several times before any real analysis of the results can be performed. Each run produces a unique result and so any effects of parameters have to be tested by repeated experimentation.

The second point is that the average chromosome length of the 10 runs is 94.8, with the maximum allowable being 100 elements. This process is known as 'bloat' and is a result of the crossover operator being able to rapidly increase the size of a chromosome. During

experimentation with the test example, it was found that the structures would increase in defining length to the maximum allowed in the run (typically 100 elements) with no improvement in fitness. The evolved equation from run 1 is shown below in reverse polish notation and shows the very long equations that are evolved using standard GP.

Result Run 1 – Generation = 499 RMS fitness = 0.137129 length = 99

```
c s - % x (1.538200) + x c c + x (-0.682093) % + + + x (2.689023) +
(1.538200) * c (7.165287) + c (-6.686818) * c c x + (6.686995) x e c x e % c
x - e c x % + + + x (6.444210) s c x e + + (-1.411281) x + (-0.682093) e c %
x (-1.411281) c % % * + (-6.630032) + (-6.993915) (2.689023) s c % * + + x
(2.166268) + x (2.689023) c c + x x (2.166268) (2.166268) - e c x x
```

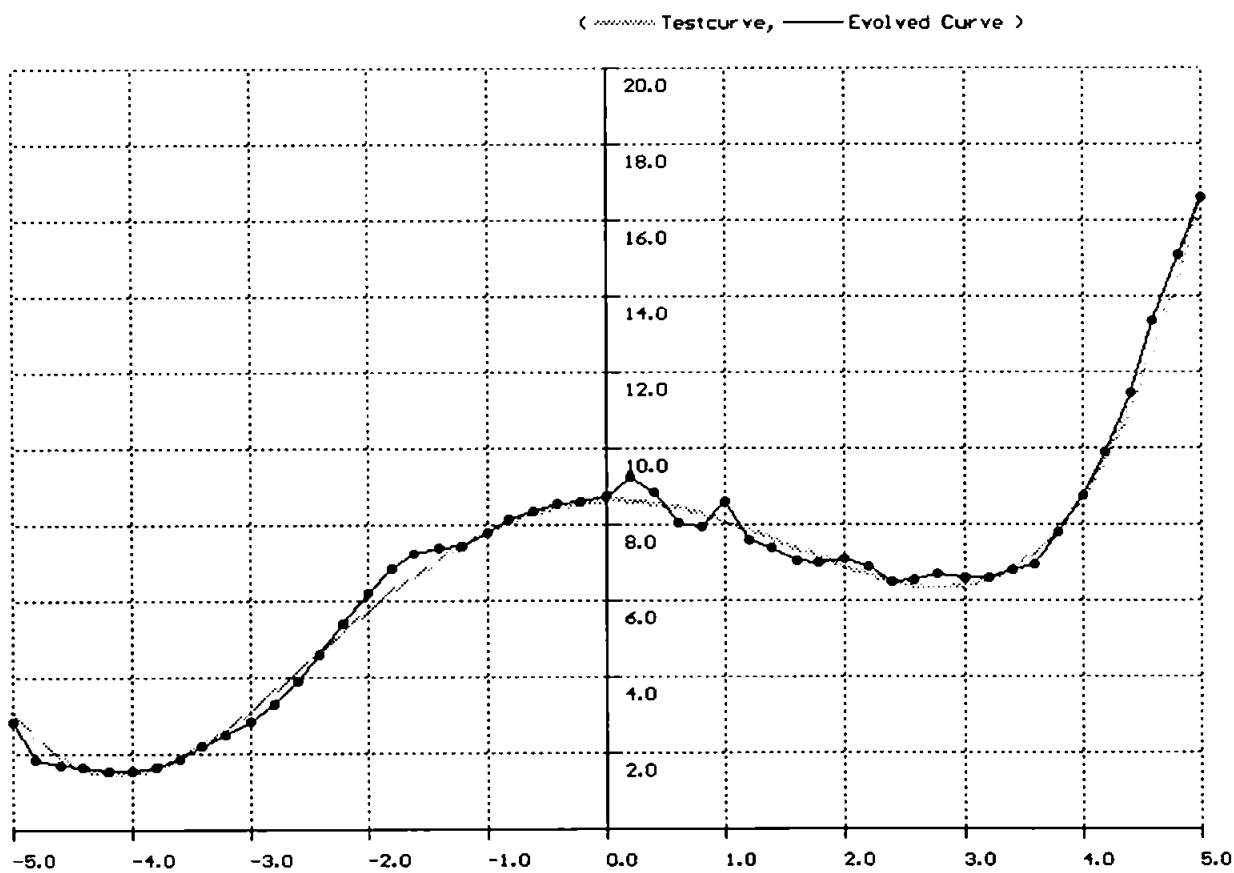


Figure 4.2 – Evolved Quartic Equation

If any one of the terminals within an individual is incorrect the associated fitness of the surface will be poor, leading to the loss of possibly good genetic material. It seems logical therefore to search through the terminal set for each S-expression to ensure that good functional information is not discarded due to poor terminal selection.

4.2 Cubic Splines

Due to the nature of cubic splines, the number of test points used can be varied. Figures 4.3 to 4.7 show the results of cubic spline fits using 3, 6, 10 and 20 points.

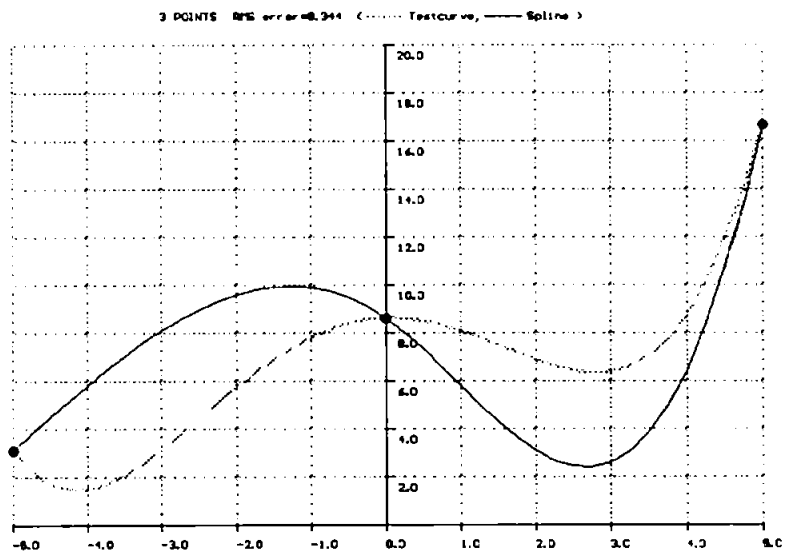


Figure 4.3 – 3 Point Cubic Spline Fit

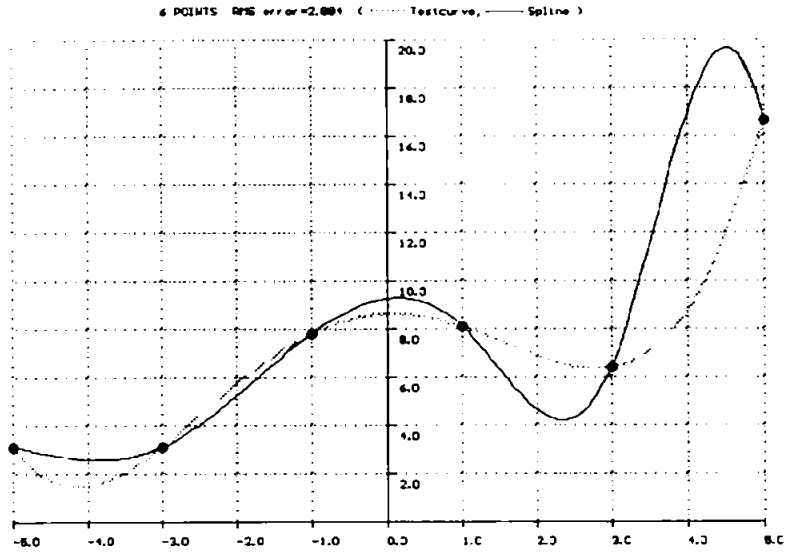


Figure 4.4 – 6 Point Cubic Spline Fit

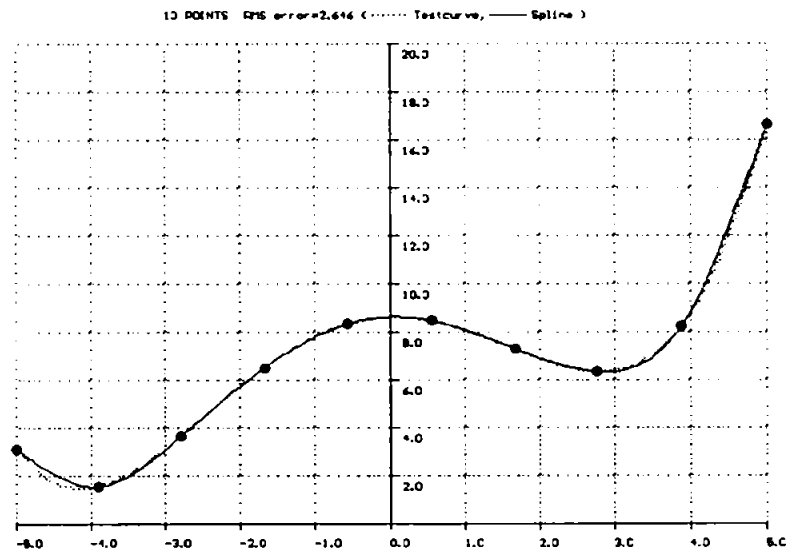


Figure 4.5 – 10 Point Cubic Spline Fit

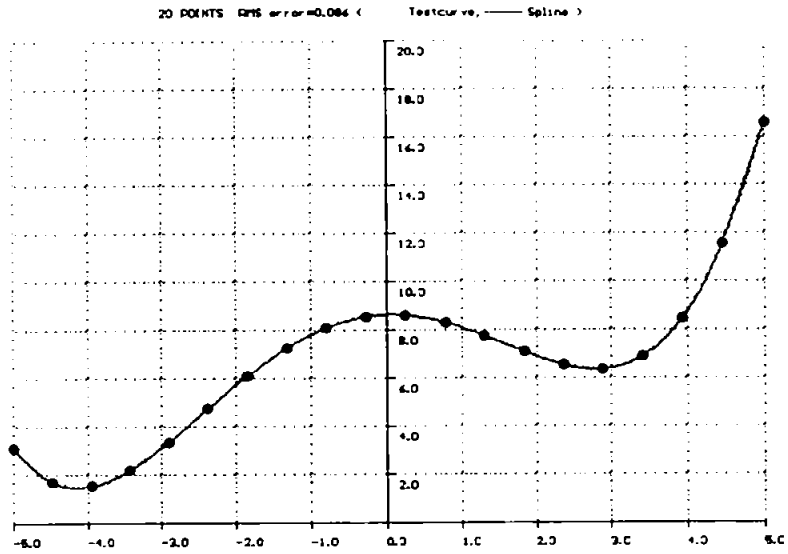


Figure 4.6 – 20 Point Cubic Spline Fit

The points used to fit the splines are taken directly from the equation to be fitted so there are no errors using this method. If however we apply the 11 fitness cases used in the GP algorithm in section 4.1, we will get a measure of the fitness of the curve fit. The RMS error is shown in table 4.2 for various numbers of points used for the cubic spline fit.

Points	RMS error
3	8.344109
4	7.763334
5	4.309230
6	2.884481
7	3.165373
8	2.645747
9	2.626185
10	2.646315
11	0.000000
12	0.108452
13	0.071587
14	0.049620
15	0.048401
20	0.086406
30	0.009566
40	0.001190
50	0.000264

Table 4.2 Results Of Cubic Spline Fit On Quartic Test Function Using 11 Test Points

These results are plotted in figure 4.7, note that the vertical axis is the log of the RMS error. The graph shows a decrease in RMS error with an increased number of cubic splines used to fit the data.

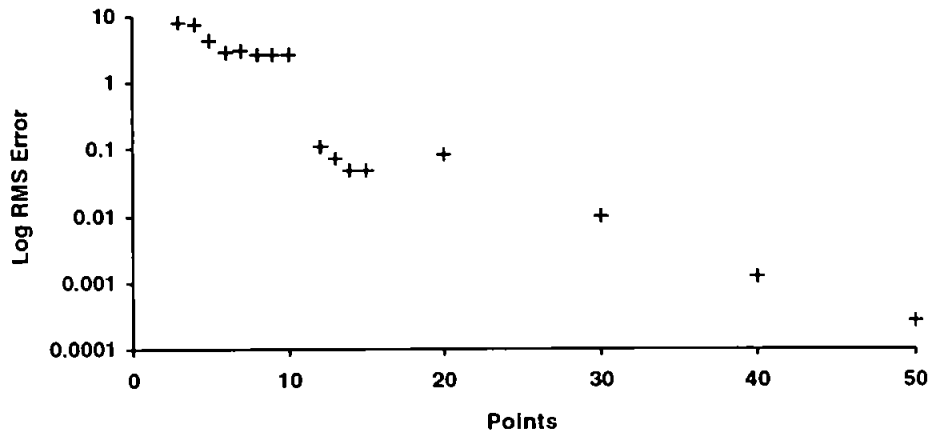


Figure 4.7 – Cubic Spline Fitting Errors

No representative mathematical function is produced which will describe all of the data in a usable way and the regression analysis is limited to 1 dependent and 1 independent variable. The final curve comprises of a series of cubic splines joined together and as such does not produce a single equation representing the data used. It is a useful technique used mainly in the field of computer-aided design and computer graphics.

4.3 Neural Networks

Neural networks are now used on two symbolic regression problems the first is the quartic test function used for the cubic splines and the second is the two-box problem (Koza,1994). The two-box problem is tested using a modified version of GP in section 5.7.1.

4.3.1 Quartic Test function

The neural network will use 11 test points, each with one input and one output. A public domain shareware package is used for the NN testing (Dannon, 1993). WinNN is a Neural Networks (NN) package which can implement feed forward multi-layered NN and uses a modified back-propagation for training.

The stopping condition for the training of the network is when all of the 11 test point are within a threshold value of 0.001. The fitness of the result is the RMS value over the 11 test points.

The network used to test the curve fitting example consists of 3 layers, an input layer, a hidden layer with 20 PEs and an output layer. The network uses a simple backprogation algorithm to adjust the weights, and the neuron function used is the sigmoid function. The learning parameters are eta=0.9 and alpha=0.9. Eta and alpha relates directly to the backpropagation learning algorithm: where the new weights are a function of the derivatives and the previous weights. Eta is the learning parameter and Alpha is the momentum. The temperature of the neuron function is a multiplier of the activation argument, in the sigmoid used here:

$$f(x,T)=1/(1+\exp(-x*T)) \qquad (4.7)$$

Changing the temperature sometimes makes the learning process faster, in most cases best results are obtained with the default value of 1, as used here.

The sigmoid function, $\sigma(x)$ is defined as: -

$$\sigma(x) = 1 / (1 + e^{-x}) \qquad (4.8)$$

The RMS Error is 0.0001499 and the network is trained after 64565 iterations.

The result of the run is shown in figure 4.8.

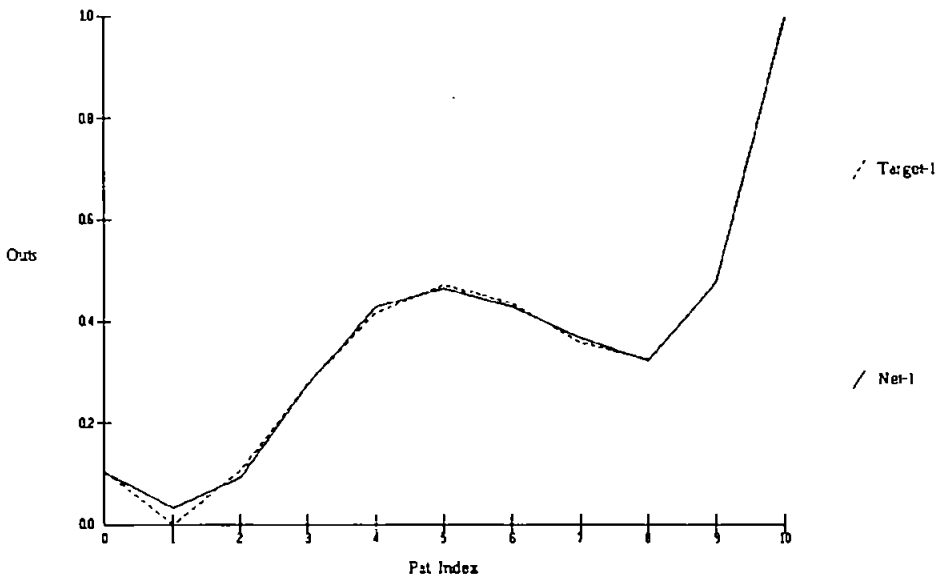


Figure 4.8 - Result of a trained neural network on the Quartic test function

4.3.2 Twobox problem

The two-box problem concerns the identification of a relationship between six independent variables (x_1 , ... , x_6), where this relationship relates to the difference y in the volumes of the first box whose length, width, and height are x_1 , x_2 , x_3 and the second box whose length, width, and height are x_4 , x_5 , x_6 (Koza, 1992).

Thus:-

$$y = (x_1\ \ x_2\ \ x_3) - (x_4\ \ x_5\ \ x_6). \qquad (4.9)$$

The goal of this symbolic regression is to derive the above equation as a "complete form" when given a set of N observations. The neural network will use 10 test points with six inputs and one output. The stopping condition for the training of the network is when all 10 test points are within a threshold vaue of 0.001. After testing the best network consisted of 4 layers, an input layer, two hidden layer with 5 PEs in each, and an output layer. The network uses a simple backprogation algorithm to adjust the weights, and the neuron function is the sigmoid function. The learning parameters are eta=0.9 and alpha=0.9.

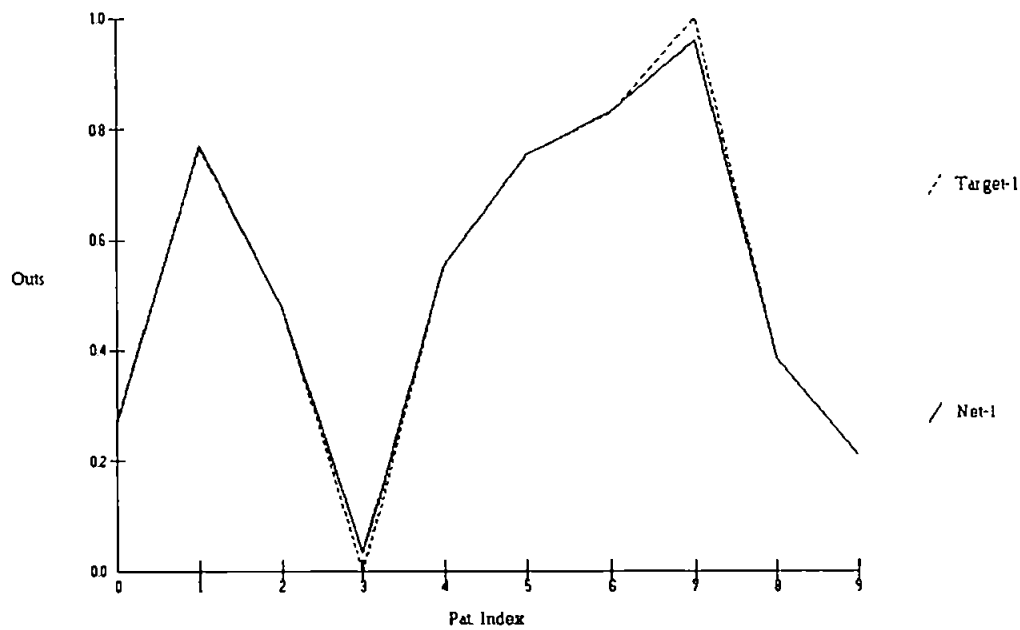


Figure 4.8 - Result of a trained neural network on the Twobox problem

The RMS Error is 0.000153193 and the network is trained after 3320 iterations. The learning parameters are eta=0.3 and alpha=0.3. Again the technique can very rapidly produce a solution to the problem. Published results (Koza, 1992) using standard GP with a population size of 4000 individuals required 1,176,000 evaluations before a correct solution is found. Further

testing of the twobox problem using DRAM-GP (Watson & Parmee, 1997) is presented in section 5.7.1. The neural network outperforms GP by a factor of over 350 times, and would be a viable method if a mathematical formula could be produced from the NN but unfortunately this is not the case

4.3.3 The Even 3 Parity Problem

The Even Parity 3 Problem (Koza, 1992, 1994) is a Boolean concept learner. The even 3 parity function *f* has 3 inputs producing a possible 2^3 outputs. The output of the 3 variables D0, D1, and D2 is shown in table 4.3.

no.	D2	D1	D0	Output *f*
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Table 4.3 - Truth Table For Parity 3 Problem

The output, *f*, takes the values 1 if the 3 input variables D0 , D1 , and D3 have even parity, i.e. an even number of them are 1. The truth table for each functional used produces a total of 4 (2^2) outputs. The neural network uses 8 input sets of three data and one output. The stopping condition for the training of the network is when all 8 outputs are within a threshold vaue of 0.00001. Through testing the network used consists of 4 layers, an input layer, two hidden layer with 5 PEs in each, and an output layer. The network uses a simple backprogation algorithm to adjust the weights, and the neuron function is the sigmoid function. The learning parameters are eta=0.5 and alpha=0.5. The result of the run is shown in figure 4.10

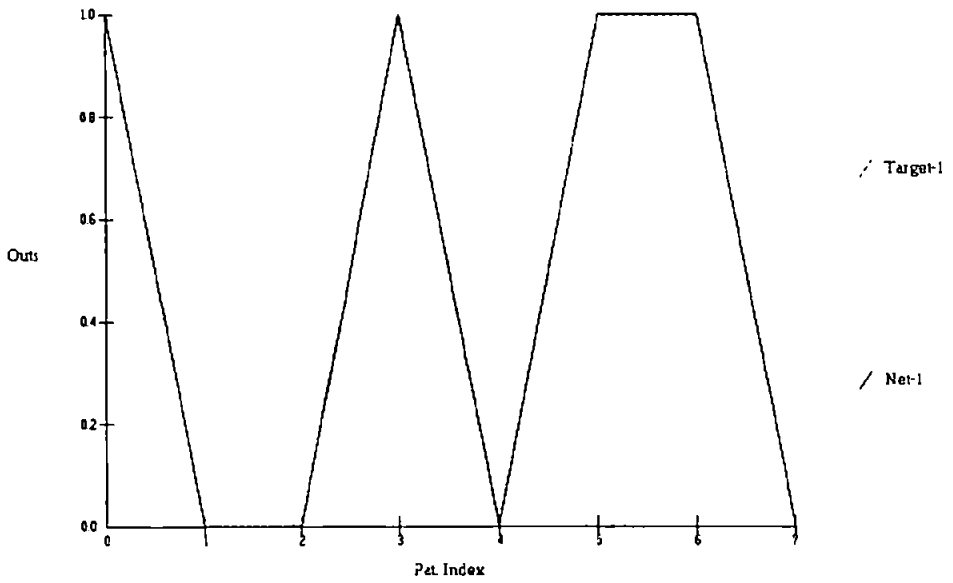


Figure 4.10 - Result Of A Trained Neural Network On The Even 3 Parity Problem

The network successfully solved the problem in 17665 iterations, with all outputs within the target error of 0.00001, with the RMS error being 0.000002688. This compares with 80,000 evaluations using standard GP with a population size of 4000 (Koza, 1992). Further testing for this problem is presented in section 5.6.1.

4.3.4 The 6-Multiplexer Problem

The input to the Boolean N-multiplexer function is the Boolean value (0 or 1) of the particular data bit that is singled out by the k address bits a_i and 2^k data bits d_i, where $N=k+2^k$. The experiments presented here have $k=2$, i.e. the 6-multiplexer. For example, if the two address bits, a_1 and a_0, are 1 and 0 respectively, the multiplexer singles out data bits d_2 (out of the 4) to be the output of the multiplexer because $10_2=2$. For an input of 100100, the output of the multiplexer is 1; for an input of 101011, the output of the multiplexer is 0. There is a total of 64

sets of data for this problem, and . table 4.4 shows the address bits, data bits and outputs for the problem.

address bits		data bits				
a_1	a_0	d_3	d_2	d_1	d_0	output
0	0	0	0	0	0	0
0	1	0	0	0	0	0
1	0	0	0	0	0	0
1	1	0	0	0	0	0
0	0	0	0	0	1	1
0	1	0	0	0	1	0
1	0	0	0	0	1	0
1	1	0	0	0	1	0
0	0	0	0	1	0	0
0	1	0	0	1	0	1
1	0	0	0	1	0	0
1	1	0	0	1	0	0
0	0	0	0	1	1	1
0	1	0	0	1	1	1
1	0	0	0	1	1	0
1	1	0	0	1	1	0
0	0	0	1	0	0	0
0	1	0	1	0	0	0
1	0	0	1	0	0	1
1	1	0	1	0	0	0
0	0	0	1	0	1	1
0	1	0	1	0	1	0
1	0	0	1	0	1	1
1	1	0	1	0	1	0
0	0	0	1	1	0	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	0	1	1	0	0
0	0	0	1	1	1	1
0	1	0	1	1	1	1
1	0	0	1	1	1	1
1	1	0	1	1	1	0

a_1	a_0	d_3	d_2	d_1	d_0	output
0	0	1	0	0	0	0
0	1	1	0	0	0	0
1	0	1	0	0	0	0
1	1	1	0	0	0	1
0	0	1	0	0	1	1
0	1	1	0	0	1	0
1	0	1	0	0	1	0
1	1	1	0	0	1	1
0	0	1	0	1	0	0
0	1	1	0	1	0	1
1	0	1	0	1	0	0
1	1	1	0	1	0	1
0	0	1	0	1	1	1
0	1	1	0	1	1	1
1	0	1	0	1	1	0
1	1	1	0	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	0	0
1	0	1	1	0	0	1
1	1	1	1	0	0	1
0	0	1	1	0	1	1
0	1	1	1	0	1	0
1	0	1	1	0	1	1
1	1	1	1	0	1	1
0	0	1	1	1	0	0
0	1	1	1	1	0	1
1	0	1	1	1	0	1
1	1	1	1	1	0	1
0	0	1	1	1	1	1
0	1	1	1	1	1	1
1	0	1	1	1	1	1
1	1	1	1	1	1	1

Table 4.4 - The Complete Data Set For The 6 Multiplexer Problem

Figure 4.11 shows the results of the run.

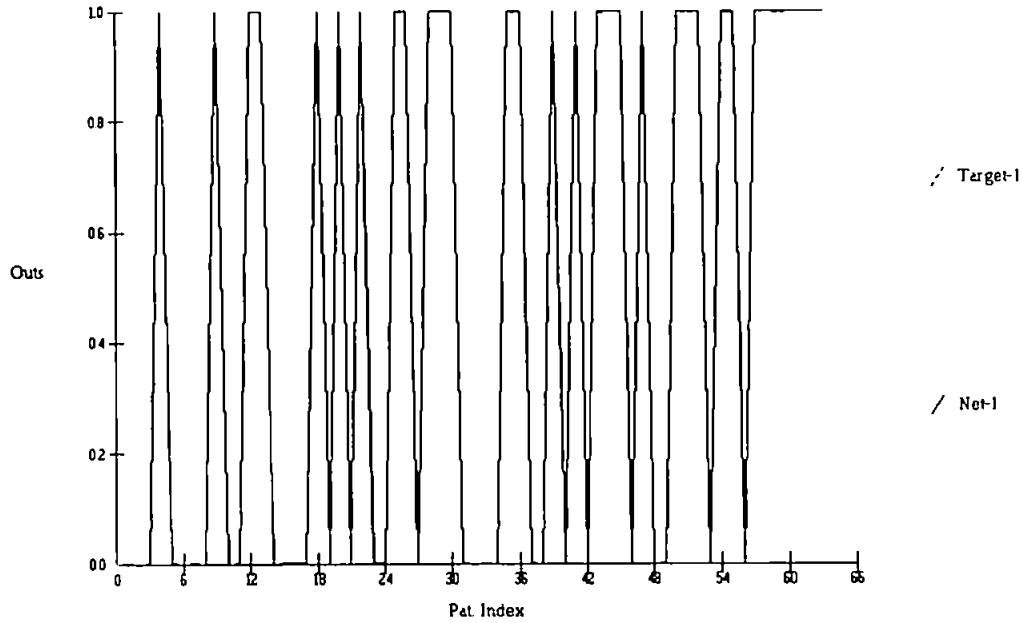


Figure 4.11 - Result Of A Trained Neural Network On The 6 Multiplexer Problem

The net solved all inputs to within a value of 0.00001 in 12440 iterations with a RMS error of 0.000001764. The network used had 4 layers (with 6,5,5,1 PE's), and the neuron function used is a sigmoid, with the learning parameters set at eta=0.5 and alpha=0.5. Section 5.6.4 presents results using DRAM-GP and standard GP solved the problem in 160,000 evaluations using a population size of 4000.

4.4 Surface Fitting

To illustrate applications to surface fitting using GP, the following test function (Lancaster, Salkauskas, 1986) shown in figure 4.12 is used to test the effectiveness of the GP paradigm.

$$z = 1, \quad \text{if } y - x \geq 0.5$$
$$z = 2(y - x), \quad \text{if } 0 \leq y - x \leq 0.5$$
$$z = 0.5\left\{\cos\left(4\pi\left[(x-3/2)^2 + (y-1/2)^2\right]^{0.5}\right)\right\}, \text{ if } \left[(x-3/2)^2 + (y-1/2)^2\right] \leq 1/16 \qquad (4.10)$$

otherwise $z = 0$.

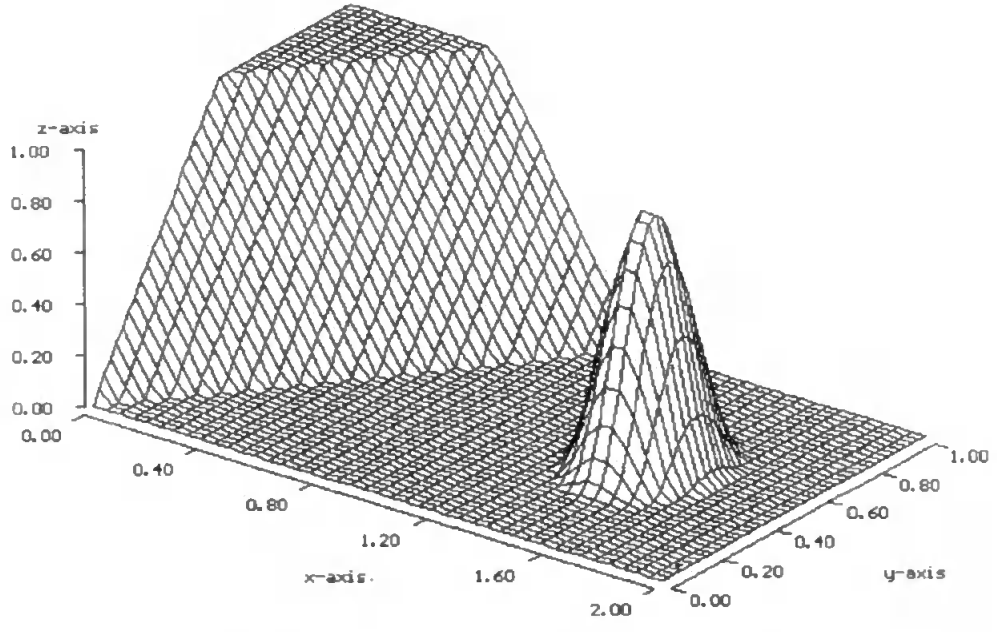


Figure 4.12 - Surface fitting test function

Surface fitting can be considered as having one dependent variable and two independent variables i.e.:-

$$Z = f(x, y) \qquad (4.11)$$

The terminal set and functional set required for GP now has to be formulated. The terminal set used for the test function is:-

$$T = \{x, y, \Re\} \tag{4.12}$$

and the functional set used is:-

$$F = \{+, -, *, \%, \sin, \cos\} \tag{4.13}$$

having associated arity 2,2,2,2,2,2 respectively.

4.4.1 The Recursive Hill Functional

Using these functionals, limited success is achieved on the test function. Surfaces which were a flat plane were produced which produced errors of around 40% when compared to the test surface.

A user-defined function describing a Gaussian type hill (or trough) with five associated arguments is thus included in the functional set. The five arguments being the mean values in x and y, the deviation in x and y and finally the maximum height of the hill. The new hill functional is of the form:-

$$z = \frac{c}{\left(\frac{(a-x)^2}{b^2} + \frac{(d-y)^2}{e^2} + 1 \right)} \tag{4.14}$$

Where a, b, c, d, and e are the arguments of the function, and:-

a = a shift in the x-axis of the hill

b = the deviation of the hill along the x-axis

c = the height of the hill

d = the shift in the y axis

and finally, e = the deviation of the hill along the y-axis.

Figure 4.13 shows the hill function.

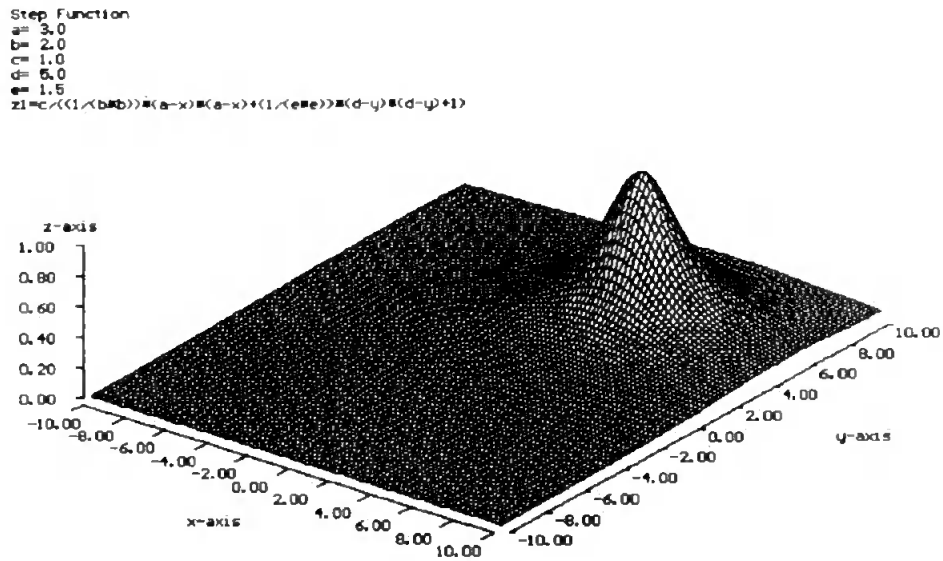


Figure 4.13 - The hill function

The reason for this additional user-defined function is to be able to compress the information required to describe the function, in much the same way that the sine and cosine functionals have been used in the curve fitting examples.

The functional set used for the test function, therefore is increased to:-

$$F = \{+,-,*,\%,\sin,\cos,hill\} \tag{4.15}$$

having associated arity 2,2,2,2,2,2,5 respectively.

It was expected that the hill functions would fit the surface under investigation by being additive

i.e. z = hill 1 + hill 2

an example of such a surface is shown in figure 4.14.

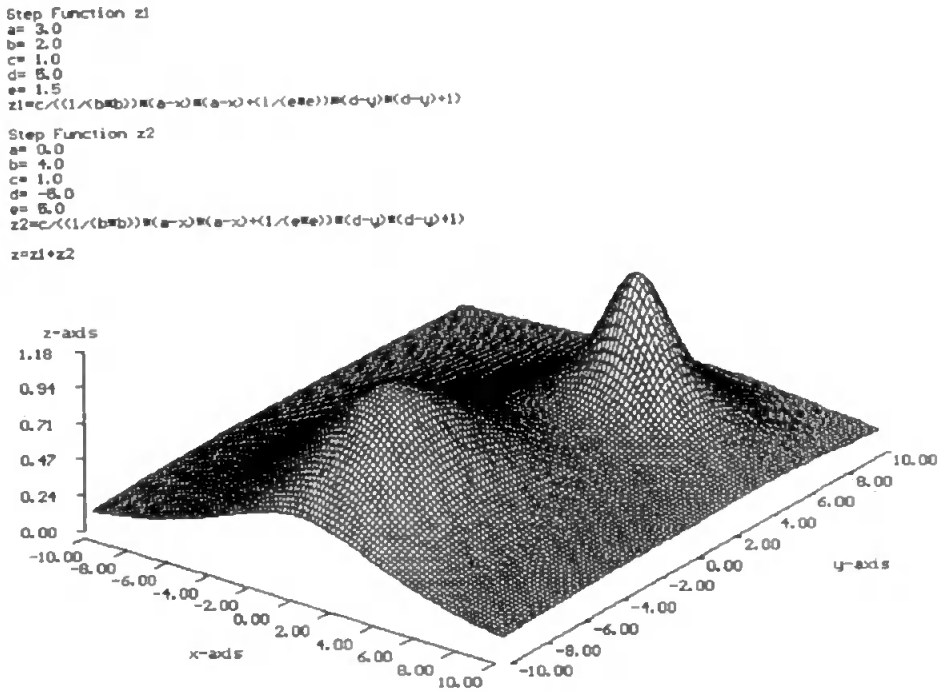


Figure 4.14 - The additive hill function

The GP started to create Recursive Hill Functions (RHF's), as shown in figure 4.15. The variety of surfaces that can be produced by these RHF's is much greater than initially thought, allowing an increase in complexity of the functions that can be evolved within the set length of the symbolic expression.

From initial runs using the test function it was found that the structures had an increased amount of terminals due to the inclusion of hill functions. If any one of the terminals within an individual is incorrect the associated fitness of the surface will be poor, leading to the loss of possibly good genetic material. It seems logical therefore to search through the terminal set for each S-expression to ensure that good functional information is not discarded. If a secondary search through possible values of the terminals is instigated, the chances of the fit functional form surviving is increased. This is a very computationally expensive technique if all individuals within the population are searched. The decision of how many individuals to be searched depends upon the computational expense associated

with the fitness function. Acceptable results have been achieved by searching the best 20 individuals within a population of 500 (the top 4%), with a maximum limit of 100 evaluations for each individual.

The terminal search is a simple hill-climbing algorithm, which loops through all terminal values and increases and decreases the value of each one in turn. The modified terminal that produces the largest improvement in fitness is then saved and the process is continued. Initially the terminals are adjusted by 10% of their original value, but if no improvement in fitness is achieved, the percentage adjustment is lowered by a factor of 2.0. Conversely, if the fitness improves and the adjustment percentage is less than 10% then it is doubled. The process continues until the adjustment percentage falls below a threshold value of 0.0001 with no improvement in fitness, or 100 evaluations are performed.

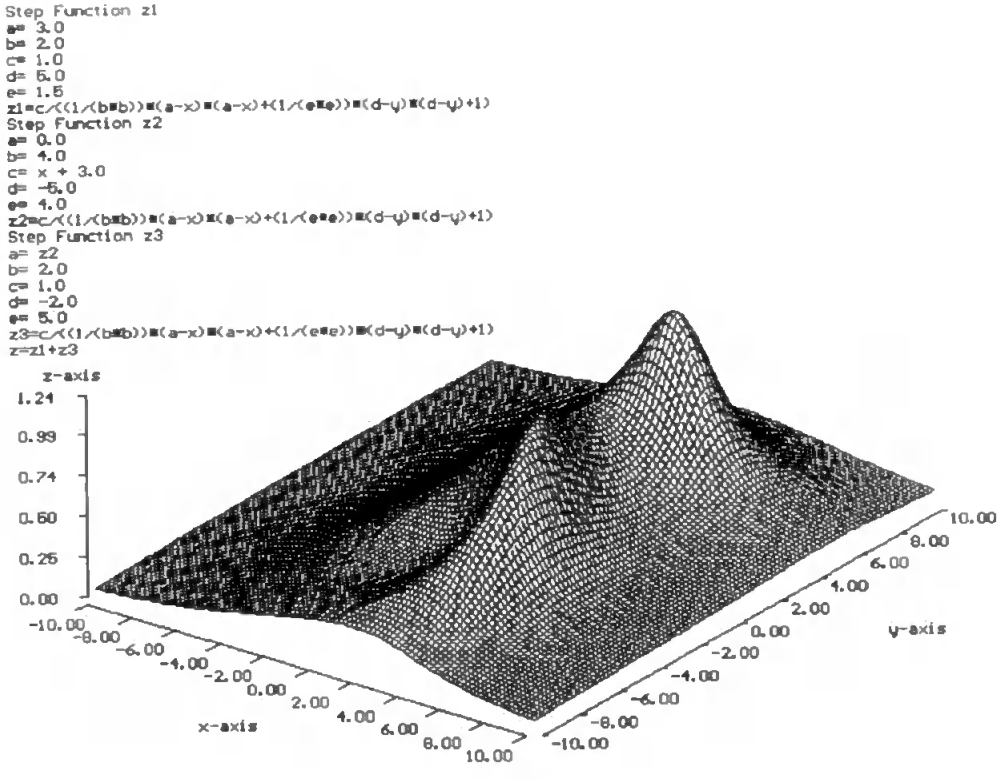


Figure 4.15 - A Recursive Hill Function (RHF)

The terminal search technique has been used to produce a surface using the test function. Table 4.5 shows the results of 2 runs without the terminal search and one run (run 3) with the terminal search.

	run 1	run 2	run 3
population size	100	500	500
chromosome length	100	100	100
pcross	0.6	0.6	0.6
pmutate	0.01	0.01	0.01
terminal search	-	-	20
test points	860	860	860
max. generations	500	500	500
defining length	29	37	50
fitness	11.834	8.445	6.885
evaluations	50,000	250,000	1,000,000

Table 4.5 - Results Of Surface Fitting Using GP

The table shows the effectiveness of the terminal search for finding fit solutions although this is as the cost of complexity of the structures (The defining length of the S-expression from run 3 is 50 elements). The improvement in fitness of the surface by using the hill-climber is achieved with a four-fold increase in evaluations required. The final surface from run 3 is shown in figure 4.16.

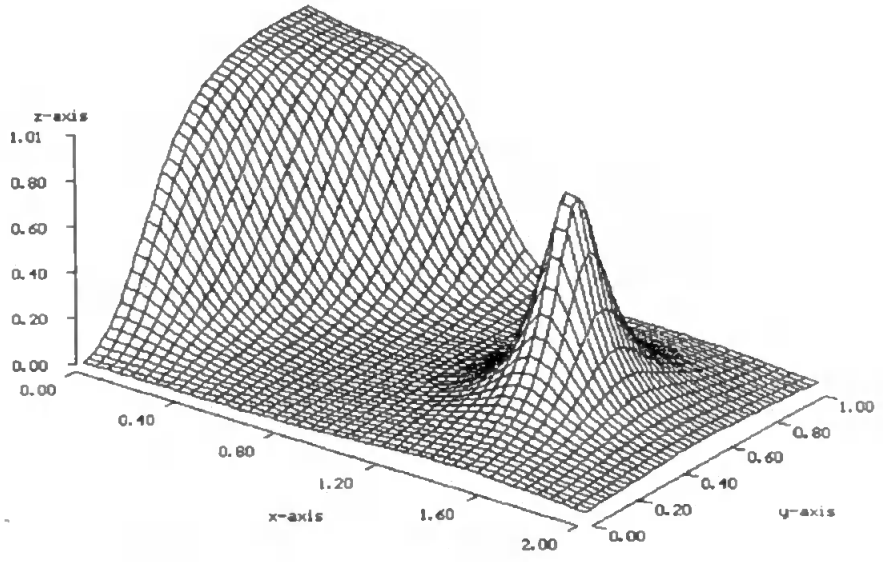


Figure 4.16 - The Evolved Test Surface Using GP

Polynomials also play a role in surface fitting where additional dimensions significantly increase complexity. Problems related to existing curve fitting techniques become more acute, and complex mathematical analysis is required to produce good results. The surface under investigation has to be represented as a series of 'patches' of the polynomial functions, normally bi-cubic patches (Lancaster & Salkauskas, 1986) and computational expense increases. Again, as with spline fitting, no useful information is derived about the system under investigation due to the piecewise solution produced and so these surface fitting techniques cannot be used for symbolic regression problems. Neural networks (Pandya, 1995) can also be used for curve fitting (as shown in section 4.3) and surface fitting and other regression problems. Design of a neural network for pattern classification may be viewed as a curve-fitting problem in hyperspace, where learning weights amounts to finding a hypersurface that provides a 'best fit' to a given set of training data. The examples presented in section 4.3 show that a NN outperforms all other techniques in terms of evaluations required. The major drawback with the method is the 'black-box' aspect, where the hidden layers of the NN prevent the user producing a usable equation. The GP paradigm shows the greatest potential for systems identification although some potential problems have been seen. The most notable is the problem of 'bloat' where the individuals increase in size up to a maximum allowed by the program. Another problem encountered is that of a suitable search of terminals for a given tree structure, the hill-climber will increase the fitness of individuals but at the cost of computational expense. The method does however produce a single mathematical equation, which represents all of the data being tested. Real-world problems are now examined using the GP paradigm in order to identify any further problems with the technique.

4.5 Modelling Engineering Systems

The previous sections were devoted to fitting either curves or surfaces to sets of data, the techniques thus developed will now be used to model 'real world' phenomenon. There are a multitude of engineering data sets, derived experimentally, which are used by the engineer in the form of graphs, look-up tables and the like. The intention here is to take this data and produce accurate models, which will describe the systems under investigation, from the data given.

Two examples of engineering systems are presented, and both are in the field of fluid dynamics. Due to the unpredictable nature of turbulent flow, many fluid problems are solved using look-up tables and graphs and so this is an ideal area in which to use evolutionary computing to attempt to produce an equation which best describes the data given. The first example attempts to find a formula for the friction factor in turbulent pipe flow. The second system involves finding general equations for the velocity vector in laminar two-dimensional flow of an incompressible fluid past a sudden expansion. This is the first time that GP (or any evolutionary technique) is to be used to solve these problems.

4.5.1 Explicit Formula For Friction Factor In Turbulent Pipe Flow

For computation of pressure drop in turbulent pipe flow an expression is required for the friction factor f as a function of Reynolds number RE and the relative roughness K/D (where K is the equivalent sandroughness of the pipe and D the diameter of the pipe). The most accurate and accepted universal formula is Colebrook and White's, where :-

$$\frac{1}{\sqrt{f}} = -2\log_{10}\left\{\frac{2.51}{RE\sqrt{f}} + \frac{K}{3.7D}\right\} \qquad (4.16)$$

This formula is implicit, that is, f appears in two places in the transcendental equation, i.e. the equation is solved by iteration, or by finding f from a graph (Moody's chart), neither of

which is convenient. Many formulae have been proposed for giving f directly for the entire range of *K/D* and *RE*. The best yet produced is probably that by S.E.Haaland (Haaland, 1983).

$$\frac{1}{\sqrt{f}} = -3.6\log_{10}\left\{\frac{6.9}{RE} + \left[\frac{K}{3.71D}\right]^{1.11}\right\} \qquad (4.17)$$

It combines reasonable simplicity with acceptable accuracy (within 1.5% of Colebrook and White's formula). The aim here is to use the GP approach to produce a solution to the Colebrook White formula, which is more accurate than Haaland's whilst still retaining Haaland's explicit nature and simplicity. The data used as the input into the GP run is calculated directly from the Colebrook White formula using the Newton-Raphson method. Due to the range of values of the friction factor and Reynolds number it was decided to set the initial functional in every individual to $\log_{10}$ this reduces the problem to finding the sub-function y in the following equation:-

$$f^{-0.5} = a.\log_{10} y \qquad (4.18)$$

where a=constant and $y = f(RE, K/D)$.

From this the functional and terminal sets can be stated.

$$F = \{ +, -, *, \% \} \qquad (4.19)$$

and $$T = \{ Real, Re, K/D \} \qquad (4.20)$$

Fitness calculation

Due to the logarithmic nature of the Colebrook White formula disproportionate errors would be introduced if a standard 'sum of the squares' fitness measure was used. For this reason the fitness of the individuals is calculated as the *sum of the percentage errors squared*.

i.e.
$$f = \sum_{i=n}^{i=0} \left[\left(\frac{x_e - x_e^*}{x_e^*} \right).100 \right]^2 \qquad (4.21)$$

where:- f = fitness

n = number of samples

x_e = evolved solution

x_e^* = Colebrook and White's solution

Table 4.6 summarises the results from a series of runs.

	run 1	run 2	run 3	Haaland's formula
population size	1000	500	500	-
chromosome length	50	25	50	-
pcross	0.6	0.6	0.6	-
pmutate	0.01	0.01	0.01	-
tsearch	-	20	20	-
test points	759	759	759	759
maxgen	75	1000	1000	-
defining length	27	9	49	11
fitness	30323	11139	10348	10067

Table 4.6 - Results Of Various Runs For Friction Factor Evolution

Using the measure of fitness mentioned above, and using 759 data points, with *RE* ranging from 3,000 to 100,000,000 and *K/D* ranging between 0 and 0.05, a fitness of 10,067 is recorded for Haaland's formula requiring a defining length of 11 (when written using the terminals and functionals used in the three runs). Table 4.6, column 2, the run 1 parameters and results shows the best result from 20 runs, and produced the following expression, in reverse polish notation:-

log10 (-4.119) - (*K/D*) % (0.6420) % (*RE*) + * (*K/D*) - % + - (*K/D*) - (0.4843) (-0.1314) (-4.2435) (*K/D*) - (*K/D*) (*RE*) (-10.8099)

(4.22)

With a defining length of 27. This can be simplified to the following:-

log10 (-4.119) - * (0.358) (*K/D*) - % * (0.642) (*K/D*) (*RE*) % (10.059) (*RE*)

(4.23)

With a defining length of 15. This expression was then used as the initial population of a second run which included the random terminal search. The best result to date from this second run is as follows:-

log10 (-3.8364) + * (0.2097) (K/D) % (11.1001) (*RE*)

(4.24)

with a fitness of 11,139. The resulting formula in standard notation is:-

$$\frac{1}{\sqrt{f}} = -3.8364 \log_{10}\left\{\frac{0.2097K}{D} + \frac{11.1001}{RE}\right\} \qquad (4.25)$$

This result is very close to the results obtained from Haaland's expression but is presented in a simpler form with no power functions (a defining length of 9 compared with 11 for Haalands formula). The accuracy of the evolved solution is within 1.8257% of Colebrook and White's formula. Solutions have been produced which give a better accuracy than the results presented in run 2 but they have large defining lengths (run 3 has a better fitness, 10348, but a defining length of 49) and thus lose the simplistic nature required of the function.

The method used to arrive at the final solution is not fully automated, and the results need to be simplified by hand before being injected into the next run. If this can be automated then runs 1 and 2 could be merged to produce one run which will take the data and finish with the fittest and shortest model for the system. This result is encouraging, the computer knows nothing about the field of fluid dynamics, but using adaptive search techniques can

successfully model a complex fluid dynamics system, using only the data of the system at work.

4.5.2 Eddy Correlation's For Laminar Two-Dimensional Sudden Expansion Flows

The problem here is that of finding a general equation for the velocity vectors in laminar two-dimensional flow of an incompressible fluid in a pipe past a sudden expansion (Badekas & Knight, 1992). At present the only method that will solve this problem is computational fluid dynamics (CFD) (Ninomiya & Onishi, 1991), and the data used to determine the fitness is derived by using CFD. No other models exist which will give the velocity at any point within the flow regime at a given Reynolds number and this is the first time that any systems identification technique has been used on this problem. Figure 4.17 shows the expansion flow model. As the Reynolds number increases the flow develops into an eddy behind the expansion which increases in length.

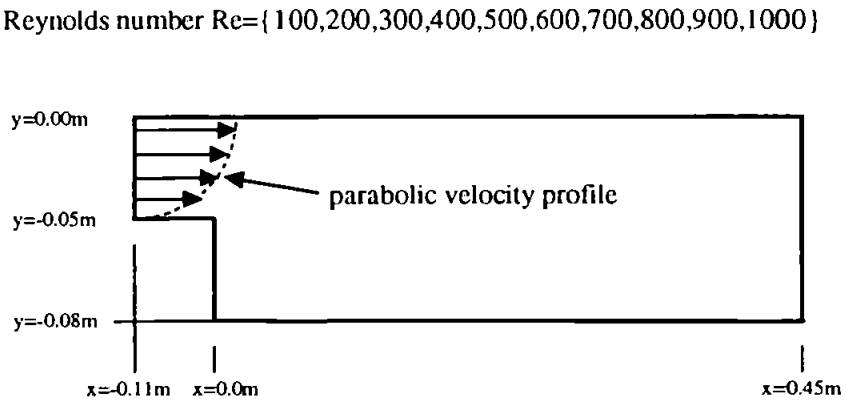


Figure 4.17 - Model for CFD expansion flow

Figure 4.18 shows the streamlines at various Reynolds numbers produced from the CFD runs which clearly show the development of the eddies.

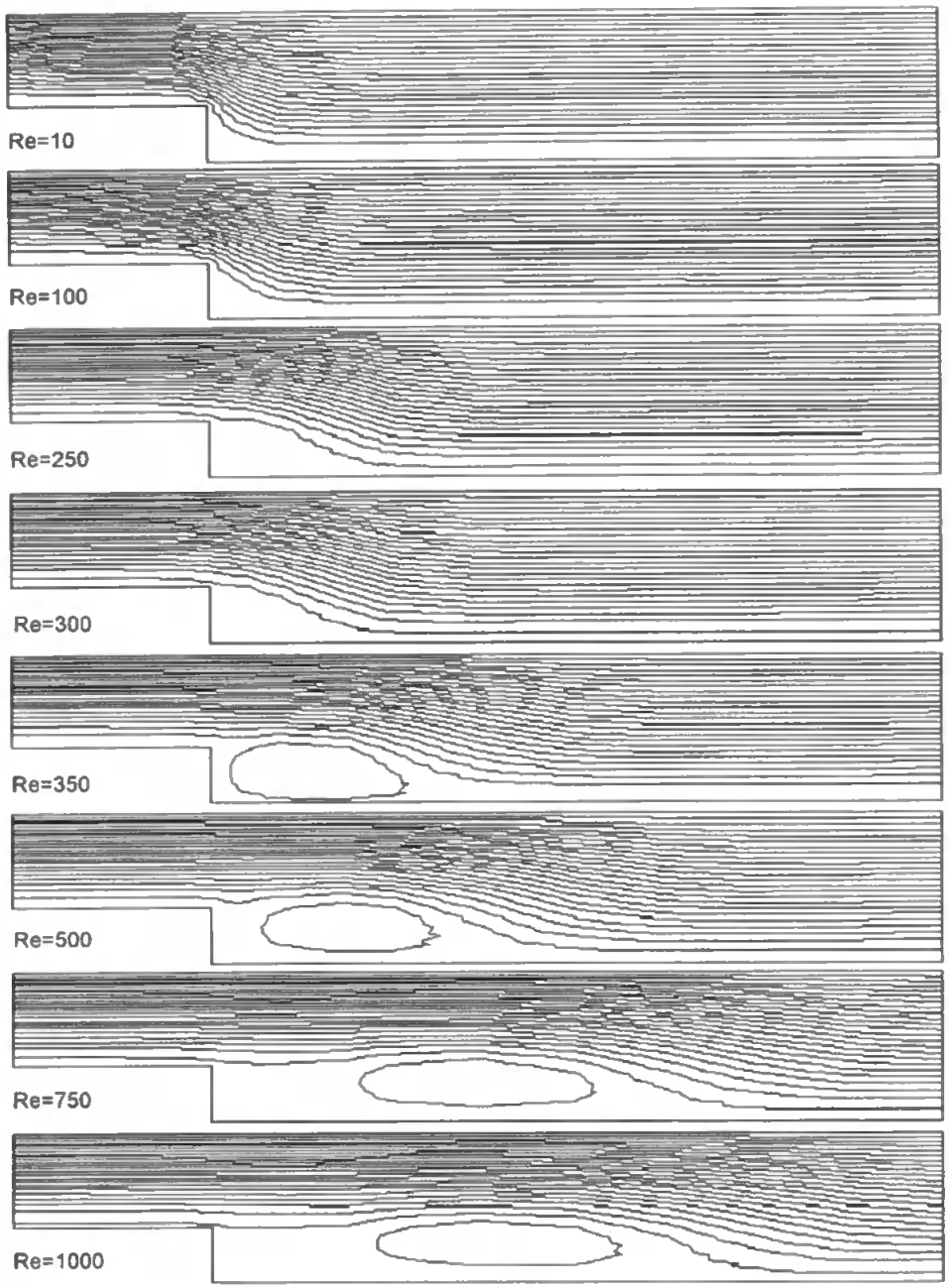


Figure 4.18 - CFD Expansion Flow Streamline Results

The CFD results produce a velocity vector and if this is divided into its x and y components the velocities can be represented as surfaces as shown in figures 4.19 and 4.20 (for Re=1000). The problem of finding a general formula for the velocity within a sudden

expansion can be produced by the fitting of these two surfaces. This decomposes the problem into simpler sub-problems (or programs) which can then be evolved.

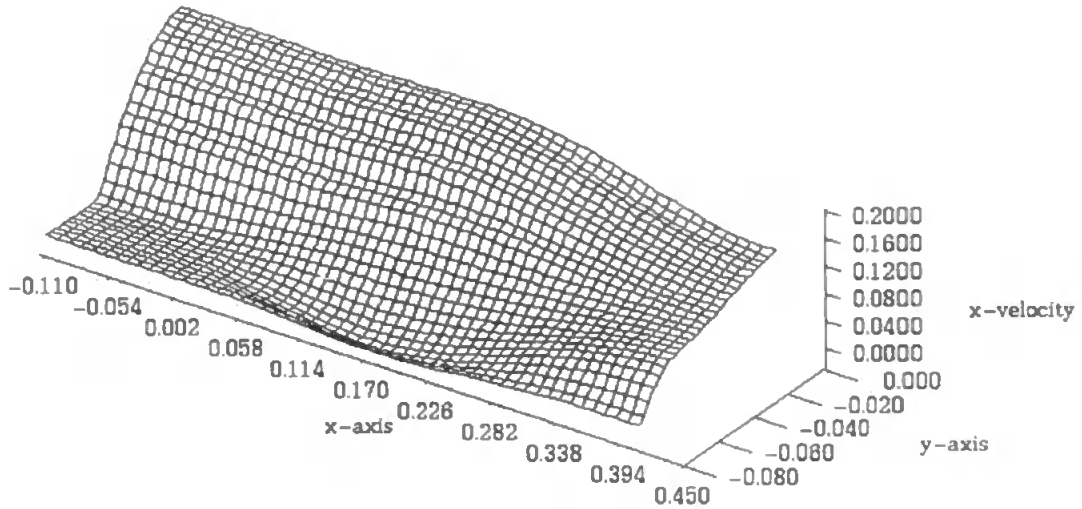


Figure 4.19 - The X-Component Of The Velocity

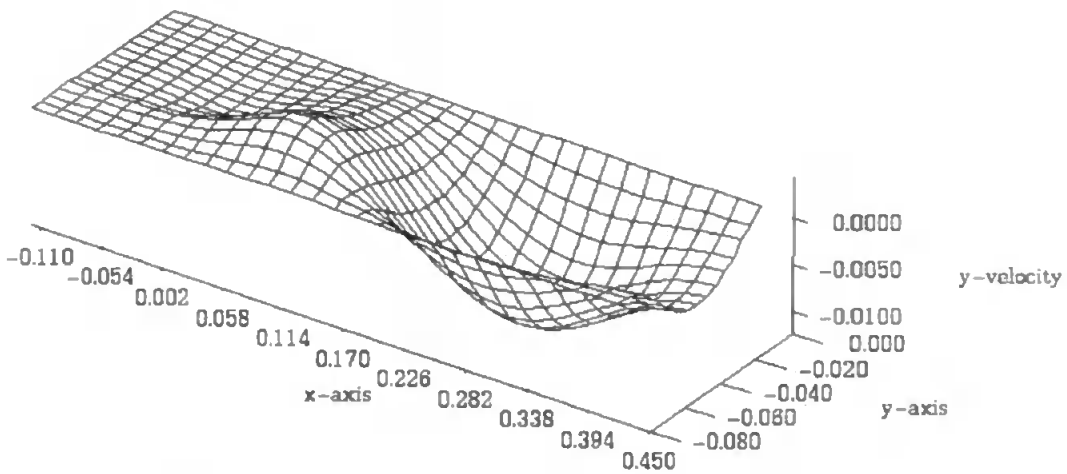


Figure 4.20 - The Y-Component Of The Velocity

The geometry of the system is set (inlet diameter and outlet diameter) and ten sets of data are produced with varying fluid velocity from Re=100 to Re=1000 in increments of 100, producing 4310 data points. Using all data points to evolve an equation for the system, with a total population size of 10,000 produced results that converged onto a flat plane through the mean of all the data points. This problem was found to be too hard for standard GP and

so initially one data set (431 points), with Re=1000, was used to find the two equations (or surfaces) for the flow of the fluid. This reduces the dimensionality of the problem.

Previous work in section 4.4 produced a set of functionals which were well suited to surface fitting, thus the terminal and functional sets for the problem are: -

$$T = \{x, y, \mathrm{Re}, \Re\} \ , F = \{+, -, *, \%, \sin, \cos, hill\} \ . \qquad (4.26)$$

The population size was set at 10,000 individuals. The evolved surfaces for Re=1000 are shown in figures 4.21 and 4.22. Table 4.7 lists the errors for Re=100, 200, …, 1000.

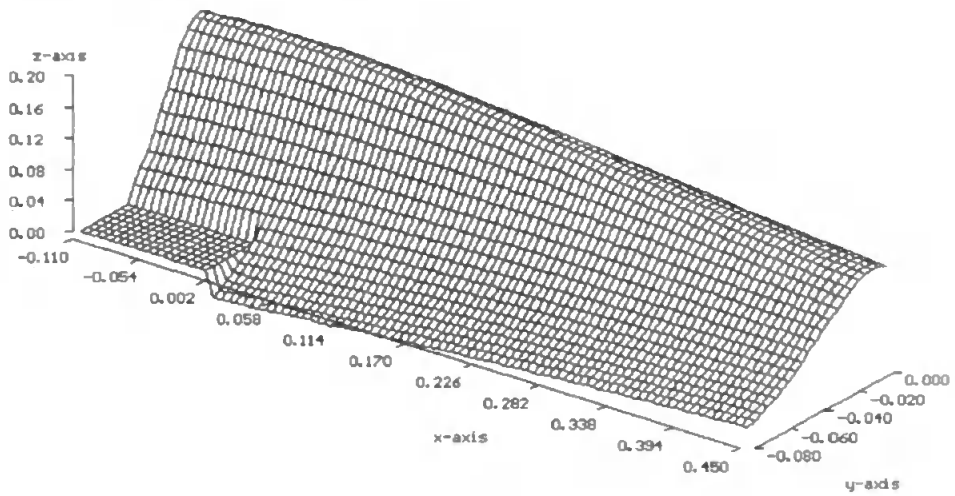


Figure 4.21 - Evolved X-Component Velocity

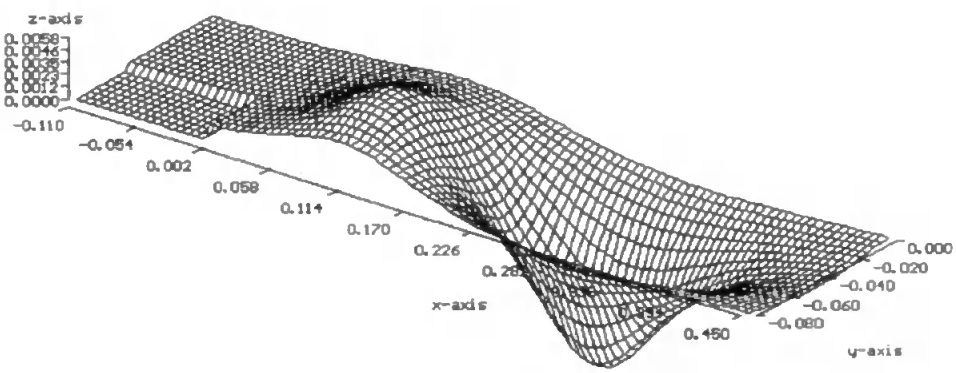


Figure 4.22 - Evolved Y-Component Velocity

The earlier work presented in section 4.5.1 showed that results from runs used to seed further runs increases the fitness of solutions, so from the results shown, a second series of runs using the results of the first run as a population seed are produced. The seeding of the population is achieved by making 50% of the population equal to the result of the first run, the remainder is then produced using the 'ramped half-and-half' method (Koza, 1992). The data used in the fitness function this time is increased and includes 5 data sets (2155 points) of Re=600,700,800,900,1000. After this run all the data is used (Re=100,200,...,1000) with the results of run 2 being used to seed the initial population of run 3 in the same way as the results from run 1 were used to seed run 2. The results of run 3 are presented and show the equations in standard notation.

Run 3 (Re=100,200,300,400,500,600,700,800,900,1000)

X-velocity Fitness=0.486538 defining length (RPN)=25

$$z1 = \frac{0.0002759\mathrm{Re} + 0.000279}{\frac{(-0.103594 - x)^2}{0.18807} + \frac{(-0.009105 - y)^2}{1.0833E-3} + 1}$$

$$z2 = \frac{\text{-}0.0015022\mathrm{Re}}{\frac{(-0349662 - x)^2}{\text{-}0.373524^2} + \frac{(-1.078300 - y)^2}{-0.475922^2} + 1}$$

X-velocity=z1+z2+0.0001540*Re;

Y-velocity Fitness=0.020050 defining length (RPN)=31

$$z1 = \frac{0.0000083\mathrm{Re}}{\frac{(0.1163 - x)^2}{4.0577\mathrm{E}-3} + \frac{(-0.0481 - y)^2}{1.3396E-3} + 1}$$

$$z2 = \frac{\text{-}0.0000175\mathrm{Re}}{\frac{(0.2661 - x)^2}{5.6400\mathrm{E}-3} + \frac{(-0.0407 - y)^2}{1.6565E-3} + 1}$$

Y-velocity=z1+z2+0.0007;

The results obtained from run 3 can be used to give the velocity vectors within the model region and can be used to show the approximate fluid flow. The evolved x and y velocity vectors are shown in figures 4.23 and 4.24 for the range Re=100 to 1000.

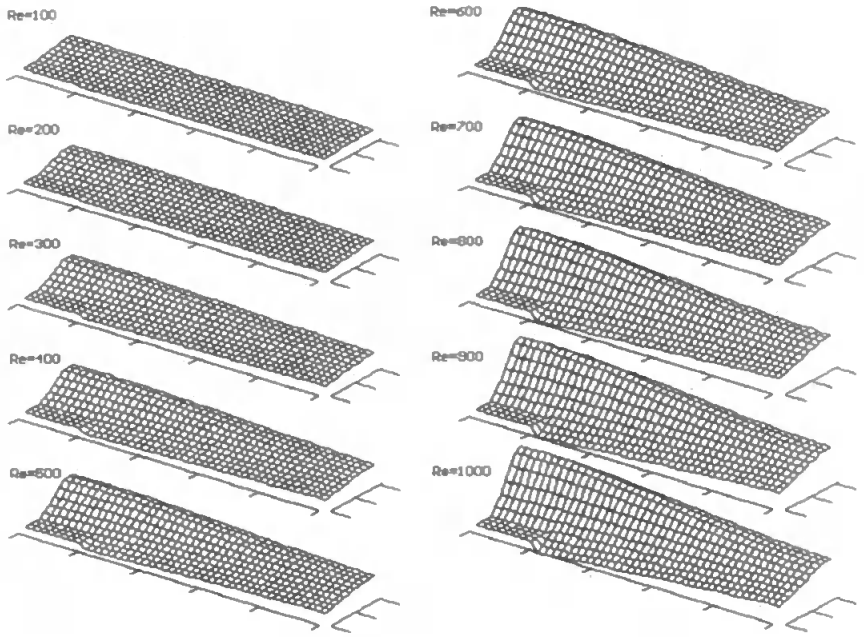


Figure 4.23 - Evolved X-Component Velocities

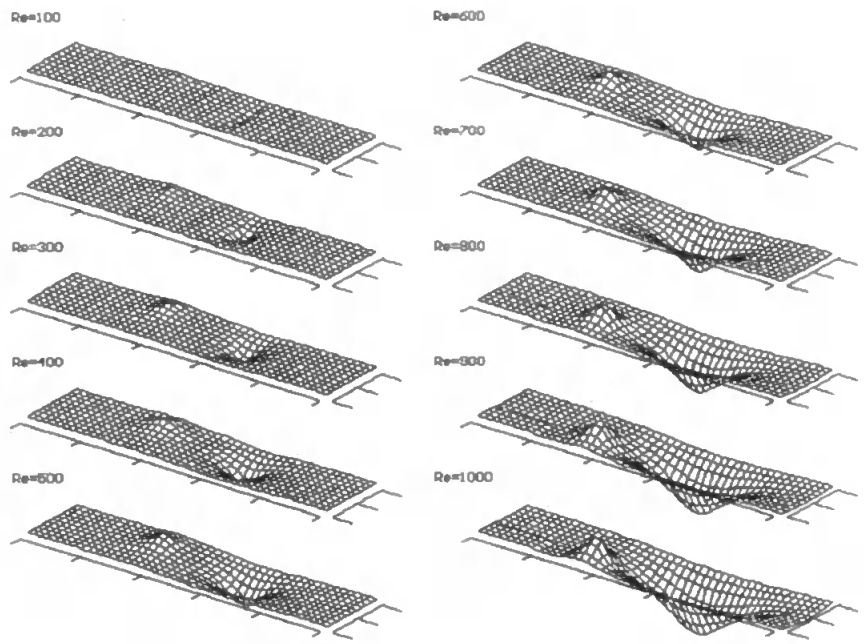


Figure 4.24 - Evolved Y-Component Velocities

The results obtained from run 3 can be used to give the velocity vectors within the model region and can be used to show the approximate fluid flow. A comparison of the errors between the CFD data and the evolved functions is presented in table 4.7.

	X-velocity	Y-velocity
Re	% error	% error
100	15.053	15.404
200	12.014	17.778
300	9.497	14.637
400	7.081	11.008
500	5.540	10.431
600	5.534	10.125
700	5.573	10.166
800	4.825	6.293
900	4.865	6.587
1000	7.379	8.157

Table 4.7- Error Comparison Of Evolved Data

The errors are due to a linear approximation of the positions of the eddy flows of the form y=mx+c, where a higher order equation could be more appropriate. As Re increases the eddy's will diverge from the approximate linear model producing the errors in table 4.7.

Close examination of the evolved flow pattern shows that at the boundaries of the system, the pipe wall, the fluid passes through the boundary. This is most apparent in the x-velocity at higher Re numbers. Figure 4.21, the x-velocity surface at Re=1000, shows errors around the step where the velocity should be zero.

To minimise the errors due to the boundary conditions of the flow model, a 'shape function' can be used. The evolved x and y velocity functions are multiplied by the shape function which then automatically defines the boundaries of the flow system, ensuring that at the boundary the x and y velocities are both zero. The shape function used for the expansion model is shown in figure 4.25. It can be seen that the function varies in height from 0.0 to 1.0 with a very steep gradient, giving the effect of masking out any unwanted errors.

Adding this function to the evolved function produced no significant change in fitness but it does prevent the flow 'moving' through the walls of the bounded system.

The shape function used to model the boundary conditions is as follows:-

a= 55.0, b= 0.0005, c=0.05, d=-3.0, e=x,

z1=c+a/(1.0+exp(d-e*(1/b)));

a=1.0, b=0.0005, c=0.0, d=-45.0-z1, e=y+0.025,

z2=c+a/(1.0+exp(d-e*(1/b)))

thus the shape function z = z2

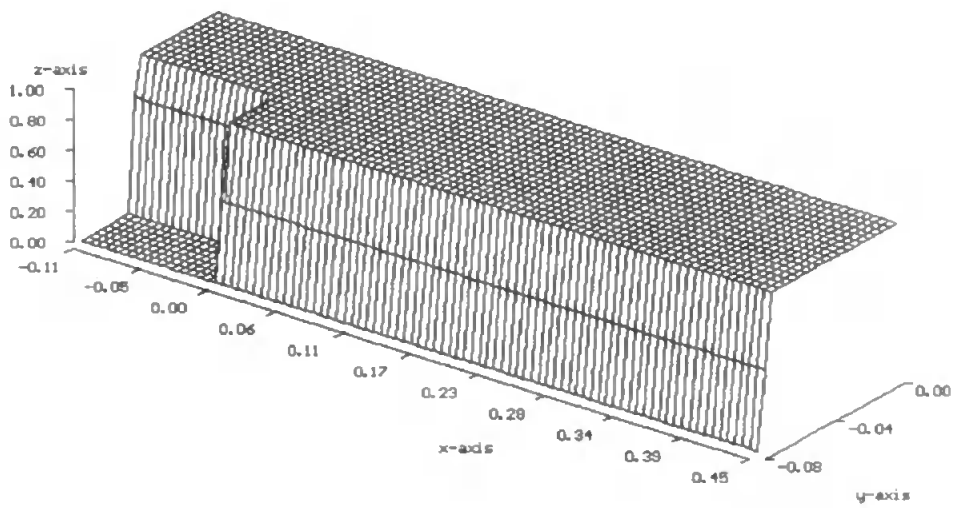


Figure 4.25 - The Shape Function Used For Fluid Boundary Conditions

Note that this shape function is a 'level 1 recursive step function'.

This function is based on the logistic curve:- $y = \frac{k}{1+e^{(a+bx)}}$. This has a horizontal asymptote y = k at infinity which is approached from below, and has one intermediate inflection point; for example, figure 4.26 shows the graph of $y = \frac{3}{\left(1+\exp^{(1-2x)}\right)}$.

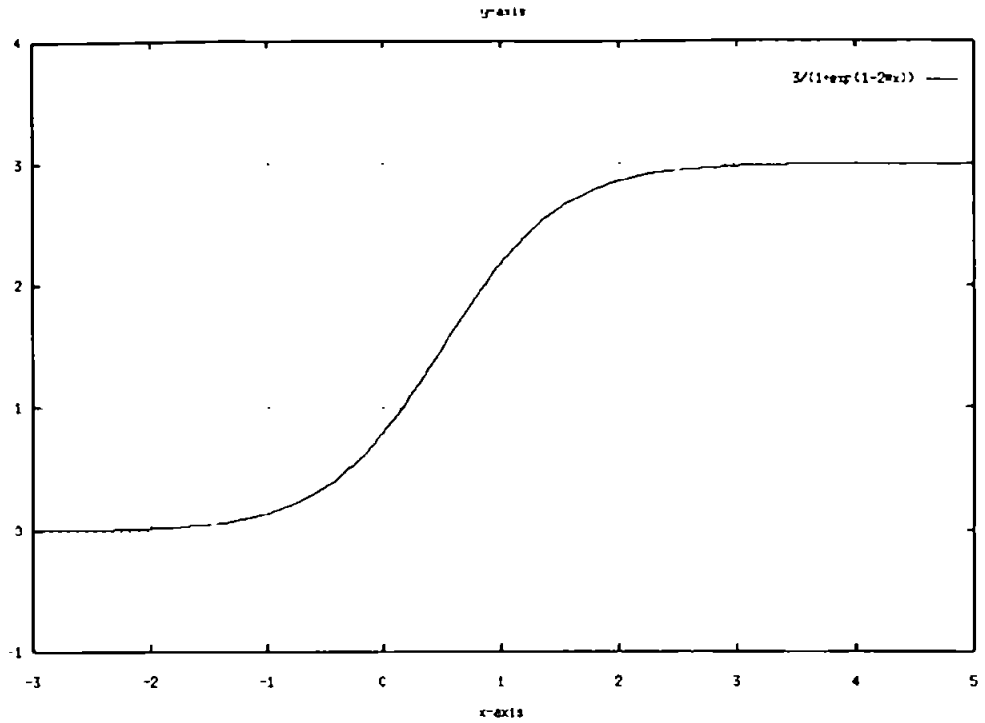


Figure 4.26 - The Logistic Curve Y = 3 / (1 + Exp (1 - 2x))

The logistic curve is frequently used to model growth in biological populations for which saturation occurs. The function is modified to the form: -

$$c + \frac{a}{\left(1.0 + \exp^{\left(d - \frac{e}{b}\right)}\right)} \qquad (4.27)$$

Where a,b,c,d and e are the arguments of the step functional, i.e. the new functional called 'step' has arity 5. The reason for the slight modification is to allow constants to shift the lower horizontal asymptote above or below zero, allowing a greater range of functions.

Using this new functional it is possible to represent complex surfaces with a minimum of elements within the S-expression. Additional functionals such as sine and cosine functionals can also be added to increase the range of surfaces that can be produced. It should be noted that the arguments for the step functional (and the hill functional) are not restricted to real

numbers, they can be composed from the set of all functions and terminals and can thus be defined as sub-trees (for example argument 'b' can be expressed as $x^2 + \sin(y)$).

Figure 4.27 shows various surfaces that can be produced from the step functional. The surfaces shown are all 'level 0 recursion' (they are composed of only one step functional, with no recursion), and are among the simplest surfaces that can be created using the step function. It is interesting to note that one of the examples is very similar to the hill functional used in earlier examples. The step functional can be considered as a primitive for hill functions, and as shown in figure 4.27, can produce many other surfaces.

This step functional could be used in addition to the hill functional for surface fitting. In the same way that the hill function can be 'recursed', the step function also has this ability to recurse and can define very complex surfaces with a minimum amount of variables required to describe the surface. Figure 4.25, the shape function used for the sudden expansion flow model, is an example of a level 1 recursive step function.

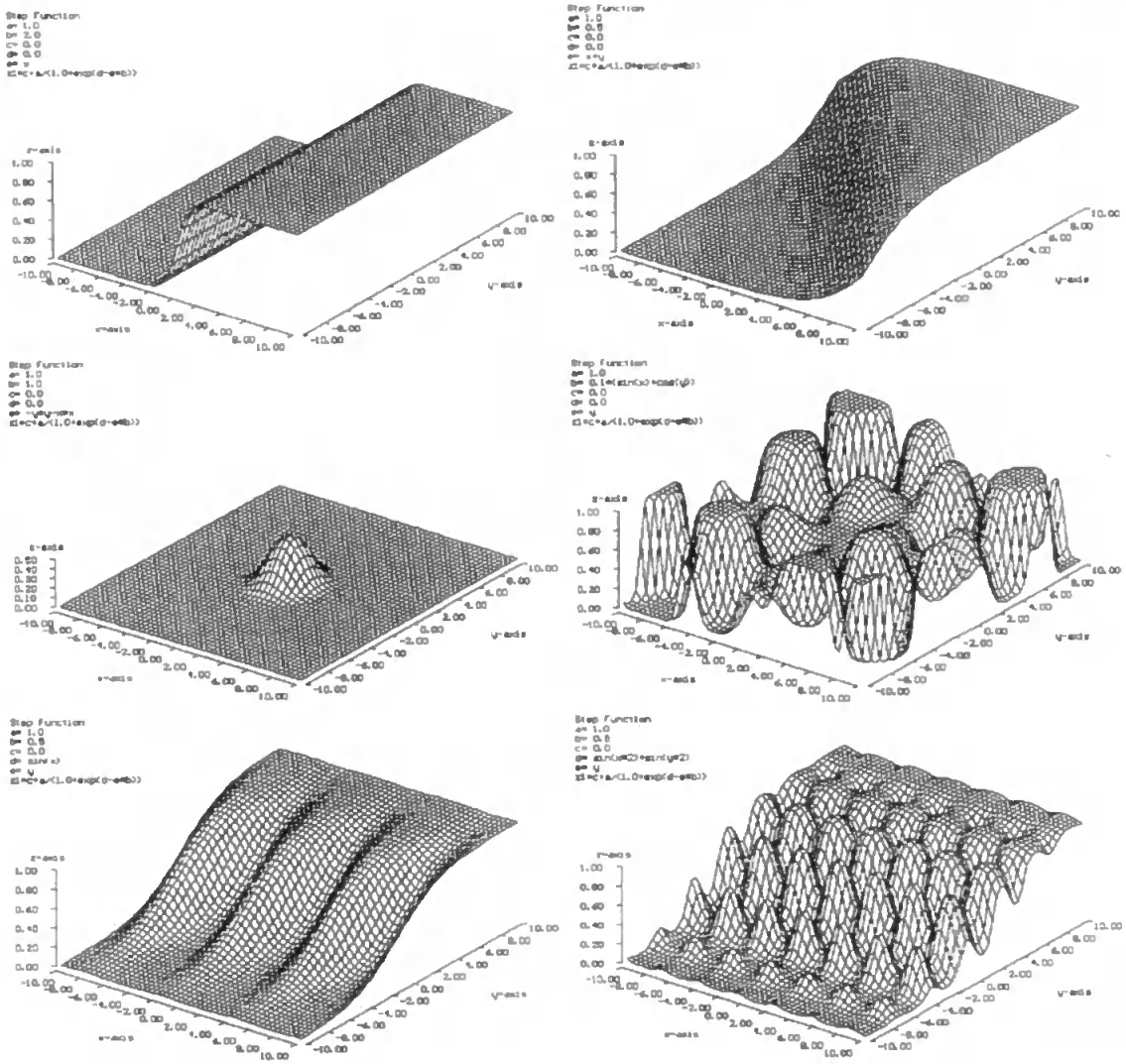


Figure 4.27 - Some Possible Surfaces Using The Step Function

Figure 4.28 shows an example of the complexity of the surface that can be produced with a minimum element length. The length of the expression required to represent the surface in RPN is 33.

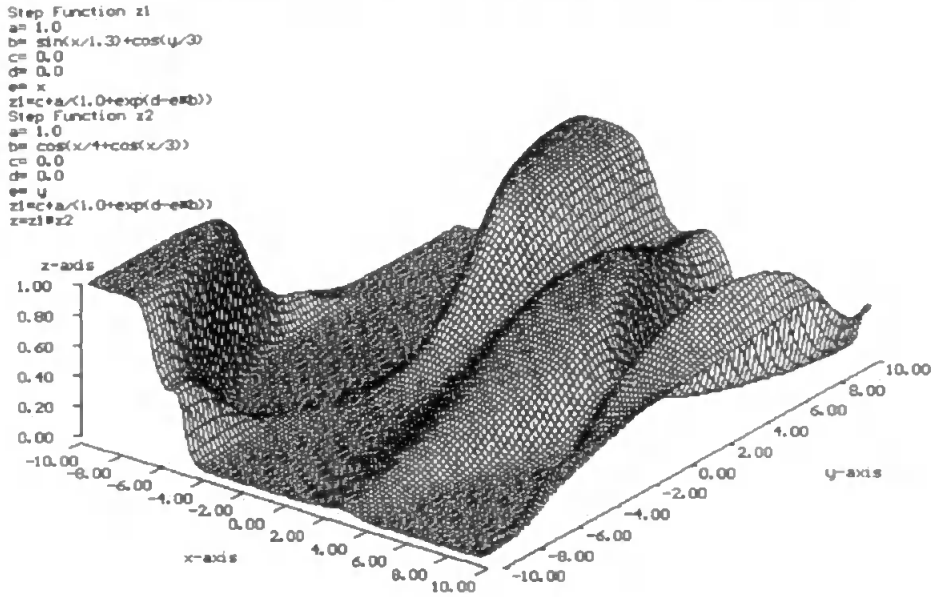


Figure 4.28 - A Complex Surface Produced Using The Step Function

4.5.3 Thermal Paint Jet Turbine Blade Data

Using the knowledge gained from the cubic test function in section 4.1, it is possible to fit empirical data of temperature cross-sections of a Rolls-Royce gas turbine blade. The turbine blade is cooled by air through cooling holes. Figure 4.29 shows a turbine blade together with a cross section showing the cooling hole geometry.

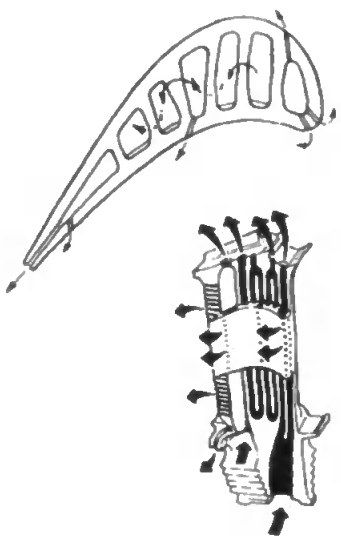


Figure 4.29 - Rolls-Royce Plc. Gas Turbine Blade

The efficiency of a jet engine can be increased by burning fuel at a higher temperature, but the first stage turbine blade has to be able to withstand this temperature. Accurate temperature readings of the blade surface are obtained by using a series of thermal paints which are designed to change colour at specific temperatures. By painting different blades with different temperature changing paints a temperature plot of the complete blade surface can be constructed.

Initial runs attempted to fit a surface to the data but only succeeded in producing flat surfaces which passed through the average temperature of the 270 test points used. The dimensionality of the problem was then reduced to that of fitting a curve to a section of the turbine blade.

The terminal set used for the thermal paint test is:-

$$T = \{x, \Re\} \tag{4.28}$$

and the functional set used is:-

$$F = \{+, -, *, \%, \sin, \cos\} \tag{4.29}$$

having associated arity 2,2,2,2,2,2 respectively.

The population size used was 10,000 with 500 generations. 20 test points were used for each blade section and the crossover rate was set to 0.6. The mutation rate was 0.001, the top 5 individuals were elite and the selection method used was roulette wheel selection.

A series of results for blade profiles is shown in figures 4.30 and 4.31.

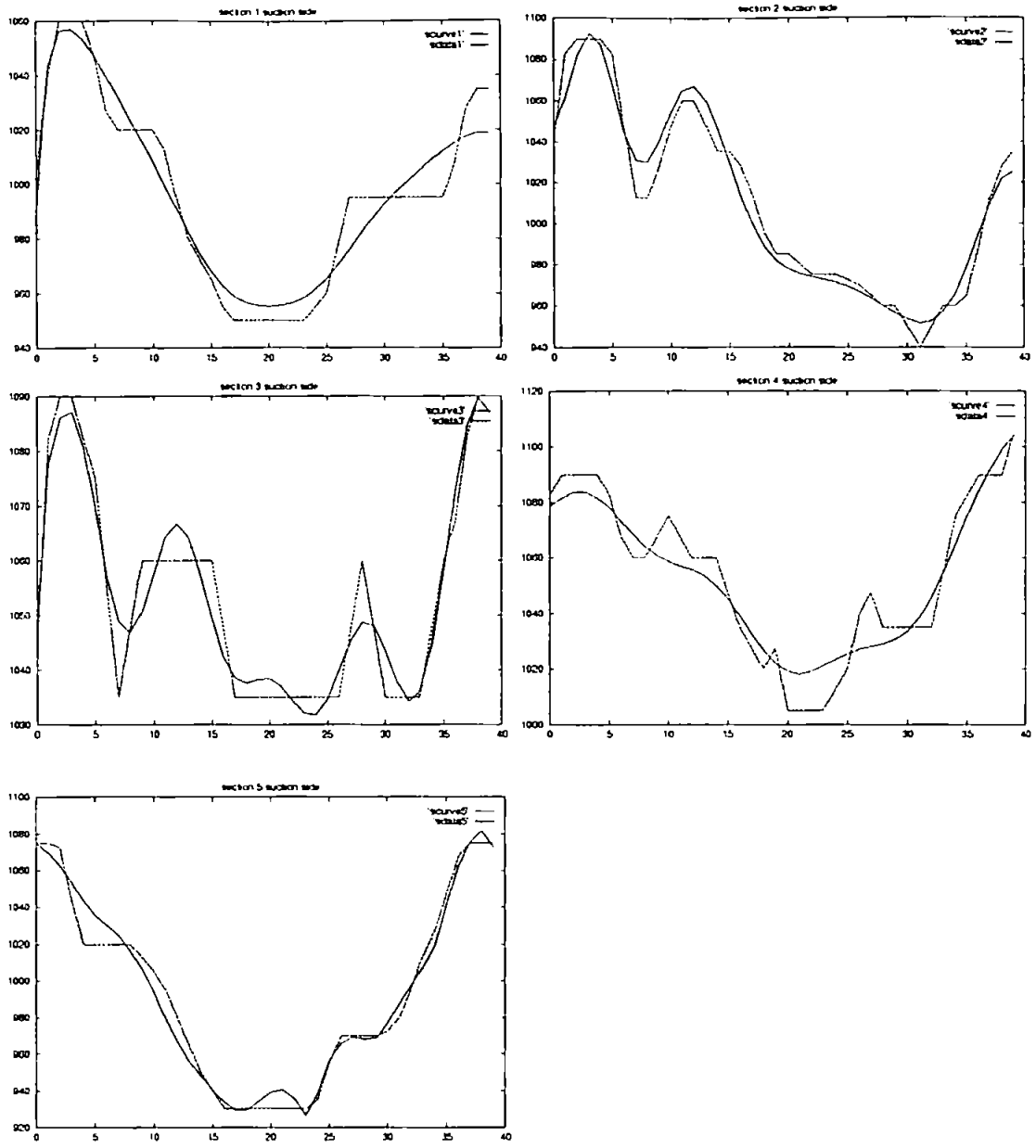


Figure 4.30 - Curve Fitting Using Rolls-Royce Plc. Gas Turbine Data

The results of one of the blade suction curve fits is shown written in reverse polish notation.

Suction side 1

```
- - + % R(460) x * R(898) R(926) * R(994) - % s * R(391) x x c % R(54)
+ % % R(545) + % R(324) x R(523) x R(523) - + * * R(84) R(54) + R(414) s
R(68) R(553) s * R(852) x
```

Where R(...) represents a real number from a list of randomly generated numbers.

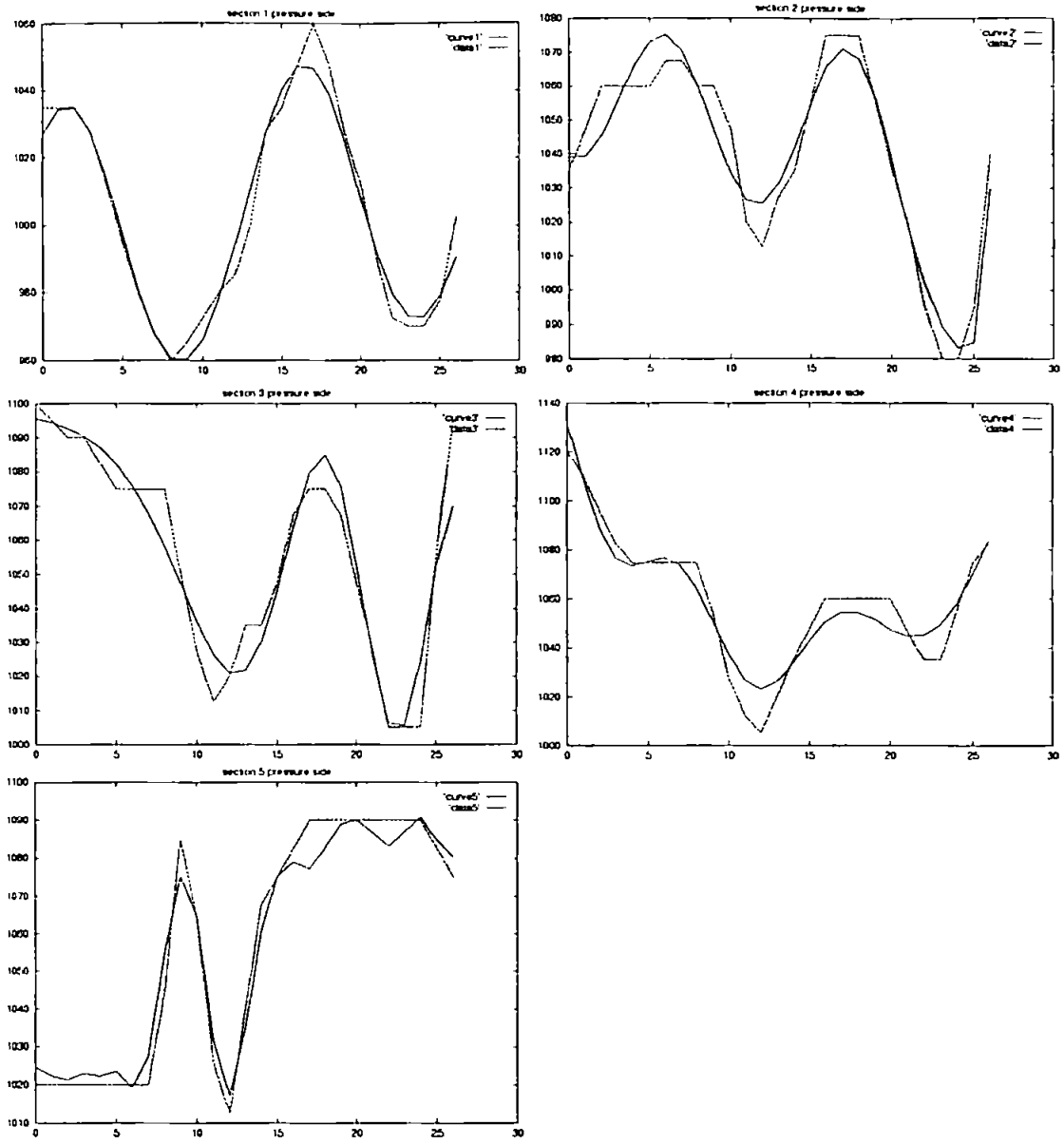


Figure 4.31 - Curve Fitting Using Rolls-Royce Plc. Gas Turbine Data

The results of one of the blade pressure curve fits is shown, written in reverse polish notation.

Pressure 1
---+xs*+R(43)R(81)x*R(352)*+R(154)R(154)s++*R(79)
xR(79)R(9)**R(925)R(931)+R(154)R(154)*+R(154)R(82)x

The GP technique replaces a subjective 'hand-fitting' method and gives a measure of the accuracy of the fit which can be compared with other section plots of the blade.

4.6 Summary

Genetic programming offers several advantages over other regression techniques, as well as certain disadvantages. The main advantage of the technique is that it provides a usable continuous equation which can be validated and used within current engineering practice. The inputs to GP are usually presented directly in terms of the observed variables of the problem domain. Therefore, the representation used by genetic programming is the natural representation of the problem domain. The lack of pre-processing is a major distinction relative to conventional genetic algorithms operating on strings, neural networks, and other machine learning algorithms.

Although neural networks can solve a wide range of problems, no direct solution is presented, and so suffers when attempting to validate the network which provides the solution to the problem under investigation. If any insight is to be gained of the system under investigation some other method must be used.

The main disadvantages of the genetic programming technique are that the solutions produced are lengthy. The equations 'bloat' to produce answers which do not provide insight into the system being looked at. The method also has limitations when solving real problems of over 2 dimensions. Additional user functionals, such as the hill function, do achieve better results but at the cost of simplicity of the results. The population sizes required to solve the problems is also very large, and as shown is of the order of 10,000 individuals.

The previous sections have shown the effectiveness of the GP paradigm for systems of increasing complexity. The progression of the work from curve and surface fitting through

to finding solutions to engineering problems has shown that the methods adopted have the potential to improve the accuracy of real world problems. However, the inherent problems of GP need to be addressed, which is the subject of the following chapter. The fluid dynamics problems are then reinvestigated using a revised GP technique to show the improvement of the new method.

CHAPTER 5

AN IMPROVED GENETIC PROGRAMMING STRATEGY

Through the work presented in chapter 4 it is apparent that GP suffers from several limitations. This chapter introduces an alternative approach to Genetic Programming, which is based upon a steady state population utilising a novel constrained complexity crossover operator. This uses node complexity weightings as a basis for dividing the population into sub-populations or species of individuals. The population is decomposed into smaller sub-populations, which communicate with each other through the action of crossover. The effectiveness of this method is demonstrated by successful application to Boolean concept formation and to symbolic regression problems and the results show that improved performance is possible with a dramatic reduction in population size and associated computer memory requirements.

5.1 Standard Genetic Programming Limitations

Two fundamental limitations of traditional GP have been reported (Iba et al, 1996), these are :-

1. Random sub-tree crossover disrupts beneficial sub-trees in tree structures.
2. No evaluation of tree descriptions. Trees can grow exponentially large or so small that they degrade search efficiency.

Traditional GP blindly combines sub-trees, by applying crossover operations. This can often disrupt beneficial sub-functions in tree structures. Thus, crossover operations seem ineffective as a means of constructing higher-order functions. Recombination operators

(such as swapping sub-trees or nodes) often cause radical changes in the semantics of the trees. This *semantic disruption* (Iba, 1996) is due to the 'context-sensitive' representation of GP trees. As a result, useful sub-trees may not be able to contribute to higher fitness values of the whole tree, and the accumulation of useful sub-functions may be disturbed. To avoid this, Koza proposes a strategy called Automatic Defining Functions (ADF's) for maintenance of useful sub-trees (Koza, 1992, 1993, 1994).

The fitness definitions used in traditional GP do not include evaluations of the tree descriptions. Without the necessary control mechanisms, trees may grow exponentially large, increasing the evaluation procedures, or so small that they degrade search efficiency. Usually the maximum depth of trees is set in order to control tree sizes, but an appropriate depth is not always known beforehand, Kinnear proposed using a size component in the fitness definition; i.e. the size of the tree is multiplied by a size factor, and the result is added to the raw fitness value (Kinnear, 1993). The use of a minimum description length (MDL) based fitness function for evaluating tree structures has been used together with a local hill-climber (Iba et. al.,1993, Iba et. al. 1996). This fitness definition involves a trade-off between certain structural details of the tree and its fitting (or classification) of errors. In order to produce an efficient guided crossover operator to search the symbolic search space a symbolic function classification is required which can then be used to minimise *semantic disruption.*

Other research relating to the manipulation of fixed design hierarchies described by both discrete and continuous variables has shown that speciation in terms of the discrete variables and the introduction of restricted crossover regimes can contribute significantly to the identification of high performance structures (Parmee 1996). Although addressing a different problem domain elements of this research have appeared relevant to the semantic

disruption problems associated with GP representations and have lead to the introduction of similar concepts described here. A Node Complexity (NC) classification has therefore been introduced which includes information concerning the complexity and lengths of the individuals. The objective is to minimise semantic disruption whilst also controlling tree length. This classification called Node Complexity includes information of the lengths of the individuals. Semantic disruption is therefore minimised whilst tree length is controlled.

Using NC separate species of solutions, classified by complexity can be established which act as discrete GP sub-populations which communicate with each other via crossover. This new approach is called **DRAM-GP** (i.e. **D**istributed, **R**apid, **A**ttenuated **M**emory, **G**enetic **P**rogramming).

5.2 Classification Of Sub-Functions

The classification of sub-functions as a guide to symbolic crossover, attempts to keep function disruption to a minimum when using symbolic crossover. The aim is to prevent large changes to the individual when undergoing crossover, producing a guided crossover operator which will also control the length of individuals.

Some possible classification methods are:-

- Dimensional analysis.
- Minimum Descriptive Length (MDL) (Iba et. al 1996). This can be applied to dimensionless parameters and is thus an improved guide to symbolic crossover. It is a trade-off between certain structural details of the tree and its fitting (or classification) of errors.
- Classification depending on computer evaluation time.

- Node Complexity weighting (NC) for each node. The NC rating is then a function of the nodes below it. The complexity will *decrease* with tree depth. Crossover is then constrained by node complexity weighting by ensuring that the child trees have *similar* NC values.

5.2.1 Dimensional Analysis

Dimensional analysis (DA) is a technique for the investigation of problems in all branches of engineering and particularly in fluid mechanics (Douglas et. al. 1985). If it is possible to identify the factors involved in a physical situation, dimensional analysis can usually establish the form of the relationship between them. Any physical situation, whether it involves a single object or a complete system, can be described in terms of a number of recognisable properties which the object or system possesses. For example, a moving object could be described in terms of its mass, length, area, volume, velocity and acceleration. Properties such as density and viscosity of the medium through which it moves would also be of importance, since they would affect its motion. These measurable properties used to describe the physical state of the body or system are known as its *dimensions*.

To complete the description of the physical situation, it is also necessary to know the magnitude of each dimension. It is not usually sufficient to know, for example, that a body has the dimension of length, the magnitude of this length is also required. For this purpose agreed *units* of measurement are used. A length would be measured in terms of a standardised unit of length, such as the metre.

In analysing any physical situation, it is necessary to decide what factors are involved and then to try to determine a quantitative relationship between them. The factors involved can often be assessed from observation or experiment. In dimensional analysis, the nature of the

factors involved in the situation is required not the numerical values. The notation adopted to indicate this is to enclose the name or symbol of the quantity in square brackets, thus [length] means the dimension of length and not a particular length with a definite numerical value. For conciseness length is abbreviated to L and the dimensions of length is written [L]. Similarly [M] is used for the dimension of mass, and [T] for the dimension of time. An equation describing a physical situation will only be true if all the terms are of the same kind and have the same dimensions. The equation is then said to be *dimensionally homogeneous*, and is valid only in relation to these dimensions. If an equation does not compare like with like, it will be physically meaningless, even though it may balance numerically. In general any equation of the form

$$a_1^{m_1} b_1^{n_1} c_1^{p_1} + a_2^{m_2} b_2^{n_2} c_2^{p_2} + \ldots = X \tag{5.1}$$

will be physically true if, in addition to being numerically correct, the terms are dimensionally the same so that

$$\left[a_1^{m_1} b_1^{n_1} c_1^{p_1}\right] = \left[a_2^{m_2} b_2^{n_2} c_2^{p_2}\right] = \ldots = X \tag{5.2}$$

where $\left[a_1^{m_1} b_1^{n_1} c_1^{p_1}\right]$ means the dimensions of $a_1^{m_1} b_1^{n_1} c_1^{p_1}$.

DA can be used to establish if a created function is dimensionally homogeneous. This reduces the search space of possible functions that will provide a solution to a given problem and thus increasing the probability of finding good solutions.

As an example suppose a formula for the volume difference between two boxes (Koza, 1994) is used.

i.e.

$$\text{volume} = a*b*c - d*e*f \tag{5.3}$$

where a, b & c are the length, width and height of box one, and d,e & f are the length, width and height of box two. The functional and terminal sets are:-

$$F=\{ +, -, *, \% \}, \qquad T=\{ a, b, c, d, e, f \}. \qquad (5.4)$$

The inputs a,b,c,d,e & f are known to have dimension:- Mass (M)=0, Length (L)=1 and Time (T)=0. The solution required is a volume (M=0, L=3, T=0). Thus expressions can be generated which have the correct dimensions (M=0, L=3, T=0) and discard *all other solutions*. The crossover operator can benefit from the use of DA, once a dimensionally homogeneous equation has been identified, crossing sub-trees *of the same dimensions* will not affecting the dimension of the complete expression. This technique reduces the search space of possible solutions and preliminary tests indicated an improvement in the search process. Its use, however is limited, primarily because dimensionless numbers are frequently used in engineering and so there are no controlling parameters for crossover within the search for a dimensionless number. This method also produces a high percentage of individuals which are not dimensionally correct which have to be discarded and is thus computational inefficient.

5.2.2 Minimum Descriptive Length

For evaluating tree structures (symbolic classification) Minimum Descriptive Length (MDL) has been successfully used together with a local hill-climber (Iba et. al. 1996). This can be applied to dimensionless parameters and is thus an improved guide to symbolic crossover. The MDL fitness is defined as:-

$$MDL = (Tree\_Coding\_Length) + (Exception\_Coding\_Length) \qquad (5.5)$$

where:-

$$Tree\_Coding\_Length = 0.5\ k\ log\ N$$

$$Exception\_Coding\_Length = 0.5\ N\ log\ S$$

Where N is the number of input-output data pairs and S is the mean square error. Crossover is controlled by choosing four parents and swapping the two worst MDL sub-trees with the two best MDL sub-trees.

5.2.3 Computing Time

Classification depending on computer evaluation time is another method that could be used to group functions. Every function produced by genetic programming will have to be tested to find the evaluation time required to evaluate the described function. On a 08486 CPU the number of internal clock cycles required for an addition of two integer numbers is 2. The number of cycles required to perform a floating point addition is between 4 and 7 cycles. The number of cycles depends upon the operator being used, the processor being used and the representation of the numbers undergoing the operations. This produces a range of possible values to use and is therefore of little use in selecting appropriate complexity values for various functions. The only way to do this accurately is to repeat the evaluation many times and average the time taken to complete the prescribed number of calculations. This has an immediate computational overhead associated with it and is therefore not considered a viable method for the classification of functions.

5.2.4 Node Complexity

Node Complexity weighting is a measure of the complexity of a tree and all its nodes. If for example a terminal set consisting of F = { + , - , * , % } is weighted (e.g. plus=1.2, minus=1.2, multiply=1.5, divide=1.5, power=2.0) each NC rating is then a function of the nodes below it and the weighting of that node. The complexity of the tree will *decrease* with tree depth. Crossover is then constrained by only crossing sub-trees with *similar* NC values. This then controls the complexity of the child trees and will provide a fitness measure for the symbolic search to aid crossover operators, as well as controlling the tree lengths.

5.3 A New Approach to Genetic Programming

The node complexity method is chosen as the bases for a new approach to GP. This method was first published at the 2nd International Conference On Genetic Programming (Watson & Parmee, 1997). The main concepts of DRAM-GP involve a steady state GP with constrained complexity crossover (CCC). Crossover is constrained by node complexity weighting values. The root node will give a complexity rating of the whole tree, and is thus used to speciate the population into smaller sub-populations. These points are discussed in detail below:-

5.3.1 Steady State GP

In the classical GP model of evolution by generation (Koza, 1992) (the generation model), each reproductive phase involves the creation of a complete new population of individuals, by selecting parents from the old population and applying genetic operations. The new population then replaces the old in one atomic step. Steady state GP has been investigated (Kinnear, 1993). The process involves evaluating an individual immediately for fitness, and then merging it into the population (or in this case a species), in place of the existing lowest fitness individual. There are no generations in steady state GP, a *generation equivalent* has passed when the number of new individuals that have been generated is equal to the population size. The population size being the *total* number of individuals (i.e. species population size times the number of species).

5.3.2 Node Complexity (NC)

NC weighting is a measure of the complexity of a tree and all of its nodes. A large tree should be assigned a high complexity value.

If for example a functional set and terminal set consisting of :

$$F = \{ +, -, *, \% \} \tag{5.6}$$

and $$T = \{ a, b, c, d, e, f \} \tag{5.7}$$

(as in the two-box problem (Koza, 1992)), the weighting of these functions is set to: - all terminals=1.0, plus=1.1, minus=1.1, multiply=1.2, divide=1.2.

Each NC value is then a function of the NC values of the nodes below it and the weighting of that node. The values chosen are heuristic, and were formulated from the following two pieces of information. Firstly, the + and – operators are less complex than the * and % operators, and as such, should be assigned smaller values. Secondly, the allocated values should be slightly larger than unity to ensure a slow increase in complexity from the terminals up to the root node.

An example of the NC weighting is shown in figure 5.1.

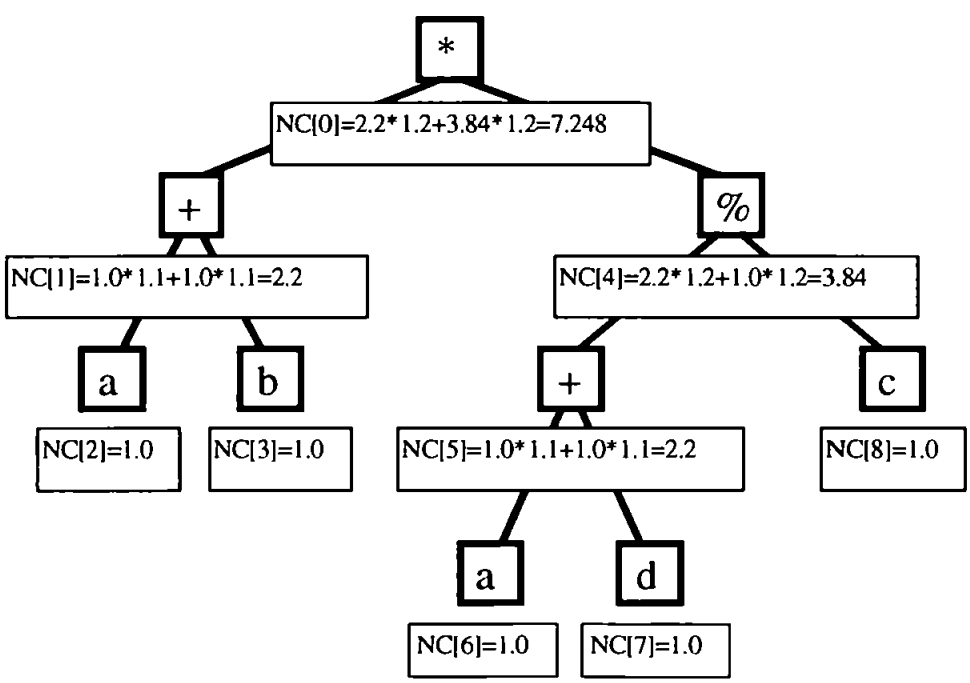


Figure 5.1 - Example Of The Node Complexity Of A Tree Structure

Each node has a specific weighting factor which is applied to the NC values below them. The NC value is then the sum of these adjusted lower node values. It can be seen that the complexity of the tree will *decrease* with tree depth, for example in figure 5.1 NC[0]=7.248 (the root node) and NC[7]=1.0 (a terminal).

Crossover is then constrained by only crossing sub-trees with *similar* NC values (from initial runs a value of ±2.0 produced the best results). This then provides a numerical complexity measure which controls crossover and minimises building block disruption by ensuring some similarity between crossed sub-trees. By only swapping sub-trees of similar complexity tree lengths are also indirectly controlled. A very small sub-tree is never replaced with a very large sub-tree and although the trees can grow in length they do not grow at the same rate as standard GP.

5.3.3 Constrained Complexity Crossover (CCC)

CCC is initiated by randomly choosing parents P1 and P2 from the total population. A cross point CP1 is randomly chosen from P1, which then defines the root node of the sub-tree to be replaced. The second cross point CP2 from P2 *MUST* then be within ± 2.0 of the NC value of CP1. The sub-tree with root node CP2 then replaces the sub-tree with root node CP1 with each allele having a probability of being mutated, C, which is initially set to 0.5. When mutating, functionals can only be mutated to other functionals, and terminals into any other terminals. This ensures that the functions created exhibit closure.

Once crossed, only one child is produced which is then evaluated and placed into the population, replacing the worst individual.

5.3.4 Species Sub-Populations

Each sub-population has a range of complexity (e.g. species type 1 where NC[0]=1.0 to 10.0, species 2 where NC[0]=10.0 to 20.0, species 3 where NC[0]=20.0 to 30.0 etc.). The two run parameters that define the species groupings is the minimum and maximum NC values. The species are then divided equally between these two limits and the population size of each species remains constant for the run. For specific values refer to the run parameters table for each problem tested. Communication between sub-populations is then achieved through the action of crossover. As a new child individual is produced it is possible that its complexity changes, and so it is placed into the correct species and evaluated.

5.3.5 Injection Mutation (I)

This mutation occurs after a set number of crossover operations (initially set to I=Population Size) and changes *only one* allele within each individual with a set probability of mutation CMUTATE. The top 5 individuals are elite and are *never* mutated, but are allowed to participate in crossover.

5.4 Performance Calculations

Before embarking on a series of tests of this process, the amount of processing required to produce a solution has to be considered. One way to measure the amount of computational resources required by genetic programming (or the conventional genetic algorithm) is to determine the number of independant runs needed to yield a success with a certian probability (usually 99% after Koza 1992,1994). Once the likely number of independent runs required is determined, it can then be multiplied by the amount of processing required for each run to get the total amount of processing required.

The amount of processing required for each run depends primarily on the product of:-

- the number of individuals M in the population
- the number of generations executed in that run, and
- the amount of processing required to measure the fitness of an individual over all the applicable fitness cases.

The process of measuring the amount of processing required is started by experimentally obtaining an estimate for the probability $\gamma(M, i)$ that a particular run with a population of size M yields, for the first time, on a specified generation I, an individual satisfying the success predicate for the problem. The experimental measurement of $\gamma(M, i)$ usually requires a substantial number of runs. Once the instantaneous probability $\gamma(M, I)$ for each generation I is known, the cumulative probability of success $P(M, I)$ for all the generations between generation 0 and generation I is calculated.

The probability of satisfying the success predicate by generation I at least once in R runs is then

$$1-[1-P(M, i)]^R \tag{5.8}$$

If we want to satisfy the success predicate with a probability of, say

$$z = 1 - \varepsilon = 99\% \tag{5.9}$$

then it must be that

$$z = 1 - [1 - P(M, i)]^R \tag{5.10}$$

The number R(z) of independent runs required to satisfy the success predicted by generation I with a probability of, say

$$z = 1 - \varepsilon = 99\% \tag{5.11}$$

depends on both z and $P(M, I)$. After taking logarithms,

$$R(z) = \log (1-z) / \log(1-P(M, I)) \quad (5.12)$$

$$R(z) = \log \varepsilon / \log(1-P(M, I)) \quad (5.13)$$

where $\varepsilon = 1-z = 0.01$ and where the square brackets indicate that R(z) is rounded up to the next highest integer. Note that P(M,i) depends on the population size M and the generation number i.

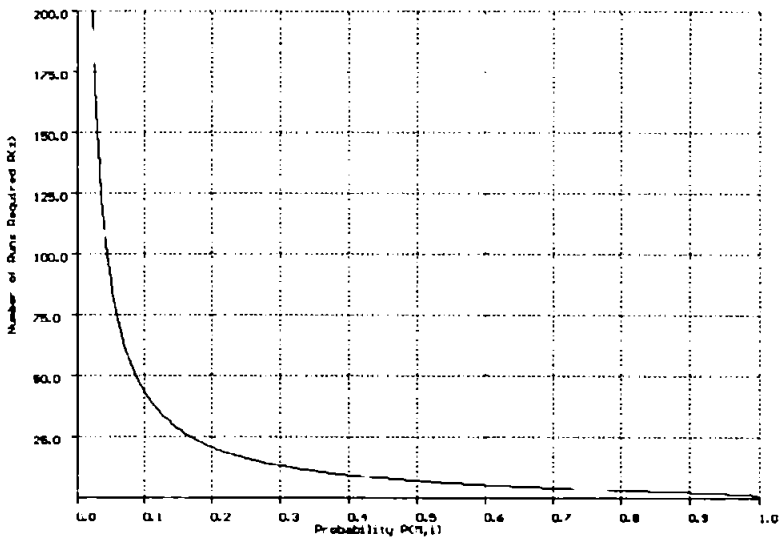


Figure 5.2 - Number Of Independent Runs R(Z) Required As A Function Of The Cumulative Probability Of Success P(M,I) For Z=99%

Figure 5.2 shows a graph of the number of independent runs R(z) required to yield a success with probability z=99% as a function of the cumulative probability of success P(M,i). For example, if the cumulative probability of success P(M,i) is only 0.09, then 48 independent runs are required to yield a success with a 99% probability. If P(M,i) is 0.68, only four independent runs are required, if P(M,i) is 0.90 only 2 independent runs are required and if P(M,i) is 0.99, only one run is required.

5.4.1 The Effect Of The Number Of Generations

The population size M and the maximum number G of generations to be run on any one run are the primary control parameters for genetic programming (as well as the conventional genetic algorithm). For a fixed population size M, the cumulative probability P(M,i) of satisfying the success predicate of a problem increases if a particular run is continued for additional generations. In principle, any point in the space of possible outcomes can eventually be reached by any genetic method if mutation is available and the run continues for a sufficiently large number of generations. However, there is a point after which the cost of extending a given run exceeds the benefit obtained from the increase in the cumulative probability of success P(M,i).

Figure 5.3 shows, for the 6-multiplexer problem (preseneted in detail in section 5.6.4), a graph between generations 0 and 200 of the cumulative probability of success P(M,i) that at least one individual in the population yields a success (i,e, the correct Boolean output for all 64 fitness cases). The graph is based on 100 runs of the problem with a population size of 20x4 (a total of 80 individuals consisting of 4 sub-populations of 20 in each) and CM1.0, IM=80.

Out of the total of 100 runs 98 were successful in finding the 100% correct solution within the 200 generations.

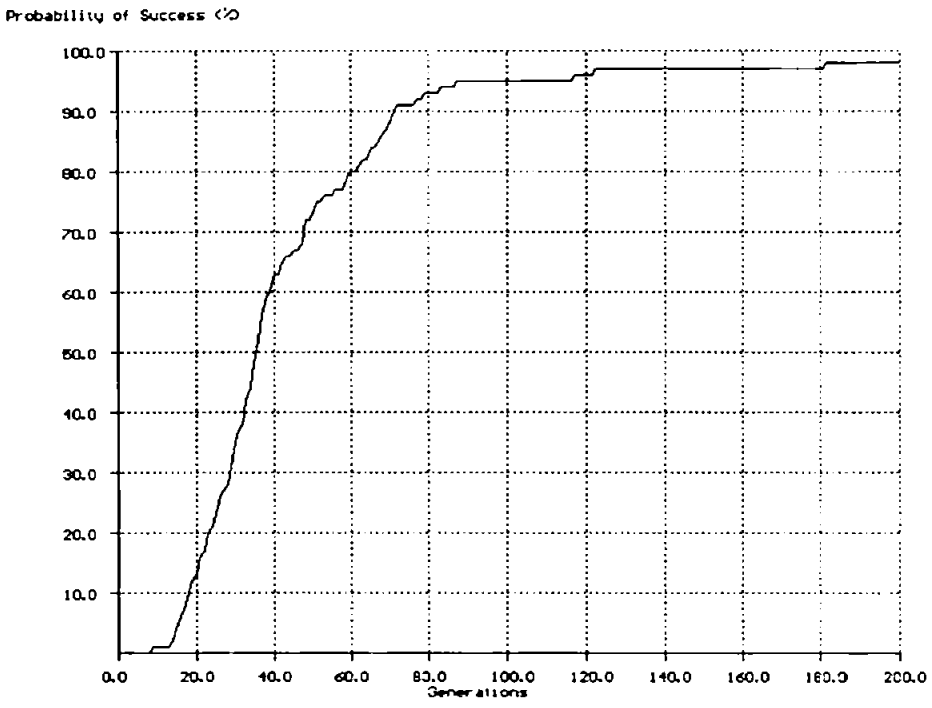


Figure 5.3 - Cumulative probability of success P(M,i) for the 6-Multiplexer problem

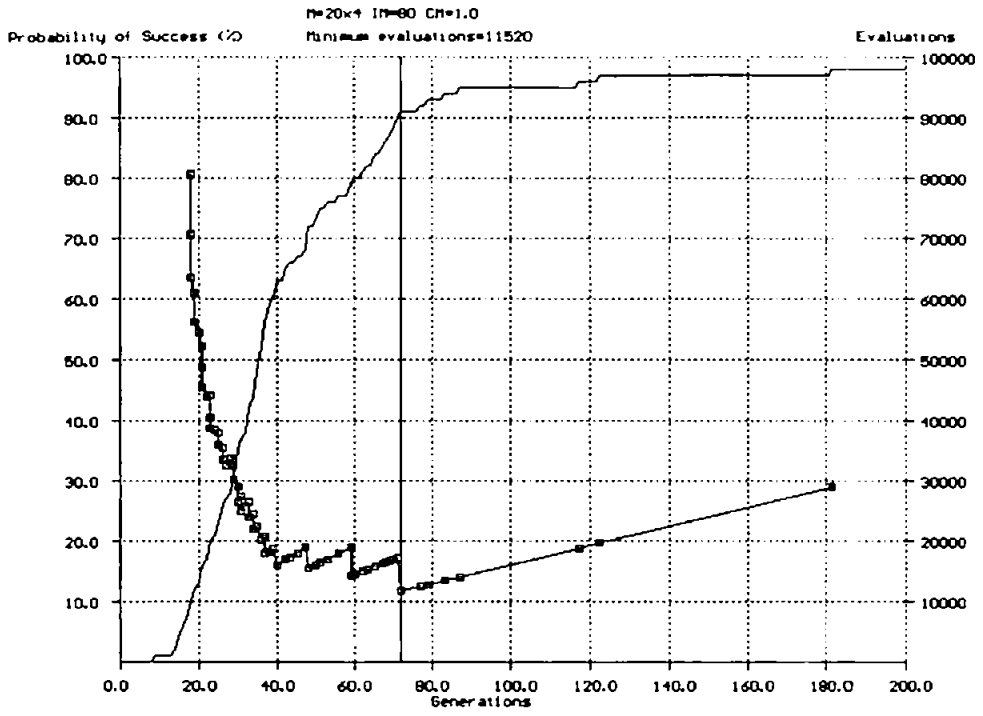


Figure 5.4 - Performance Curves For The 6-Multiplexer Problem

Figure 5.4 shows the evaluations required reach a minimum at generation 72 with a total of 11,520 individuals needing to be processed to achieve a probability of success of 99%. This is indicated by the vertical line at 72 generations.

This performance measure requires many runs using the same parameters to achieve accurate results. All results presented within this thesis are produced from 100 runs on the problem.

5.5 Testing The New Paradigm

Boolean concept learning (or Boolean induction) is an important part of machine learning, and can be regarded as a type of pattern recognition, in which the input (independent) and output (dependent) variables are binary. The effectiveness of DRAM-GP is initially demonstrated through 4 experiments.

5.6.1 The Even Parity 3 Problem

To test the effectiveness of the new genetic programming strategy as a Boolean concept learner, a simple experiment called "parity 3" (Koza, 1992, 1994) will be tested. The even 3 parity function *f* has 3 inputs producing a possible 2^3 outputs. The output, *f*, takes the values 1 if the 3 input variables D0 , D1 , and D3 have even parity, i.e. an even number of them are 1.

The Functional Set

The Functional set used for this problem is:- F = {and, or, nand, nor} with arguments {2,2,2,2} respectivly. This function set is computationally complete and is sufficient to solve any problem of symbolic regression involving Boolean functions.

The truth table for each functional used produces a total of 4 (2^2) outputs the number of true outputs is listed in table 5.1, and these results are used to decide on the node complexity of each functional.

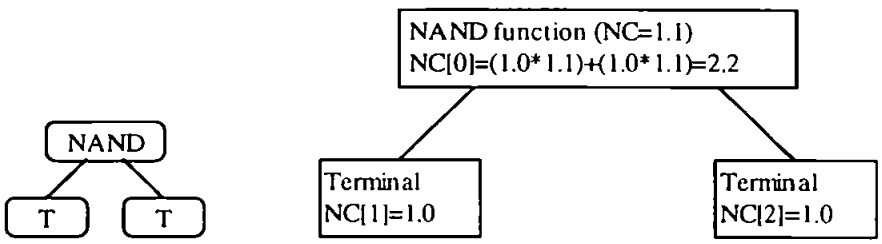
function	no. of true outputs	Node Complexity
AND	1	1.3
OR	2	1.2
NAND	3	1.1
NOR	2	1.2

Table 5.1 - True Outputs And Node Complexity Values For All Functionals

The Node Complexity weightings for the functionals were chosen based upon the number of output values that are TRUE for each functional. The less the number of TRUE outputs the more complex the function. The NAND function will produce 3 TRUE outputs of the possible 4 and has node complexity of 1.1, the OR and NOR functions each have 2 true outputs and are assigned a value of 1.2, whilst the AND functional has a NC value of 1.3

due to only having 1 of a possible 4 true values. The terminal set consists of the inputs D0, D1 and D3 which are all assigned node complexity values of 1.0.

The maximum and minimum root node complexities of each individual now need to be considered. The selection of the first label is restricted to the set of functions because a hierarchical structure is required, not a degenerate structure consisting of a single terminal. With this in mind the simplest individual created will have NC[0] = 2.2. This is illustrated in figure 5.5. The individual has a chromosome length of 3, consisting of the functional NAND and two terminals. Since all terminals have the same node complexity it is not necessary to specify which terminal is used for the purpose of calculating possible complexity ranges.



5.5 (a) **5.5(b)**

Figure 5.5 - The Simplest Individual That Can Be Created Of Length 3

The individual is shown in figure 5.5(a) and consists of 1 internal node (a function F) and two external nodes (terminals T). Table 5.2 shows the NC[0] values for all of the possible individuals of length 3.

Function (RPN)			NC[0]
AND	T	T	2.6
OR	T	T	2.4
NAND	T	T	2.2
NOR	T	T	2.4

Table 5.2 - The Root Node Complexities For All Individuals Of Length 3

No individuals of length 4 can be created due to all the functionals having 2 arguments. Individuals of length 5 can have two possible structures, where F is a functional and T is a terminal, these are shown in figure 5.6.

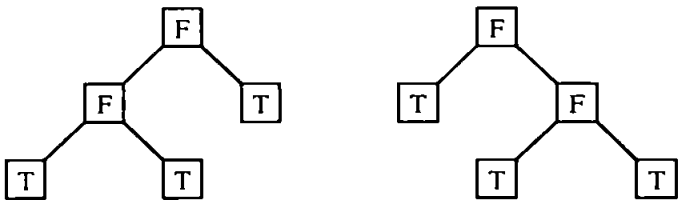


Figure 5.6 - The Two Possible Structures For An Individual Of Length 5

The number of internal nodes is now 2 with 3 external nodes. Using the available functionals, 32 possible NC[0] values can be created. The two structures will produce the same root node complexity if the internal nodes appear in the reverse polish notation in the same order. The 16 NC[0] complexities that can be created are listed in table 5.3. For example, if the first row of table 5.3 is examined , a program structure is shown composed of:-

AND AND TERMINAL TERMINAL TERMINAL

This is shown in reverse polish notation and the associated tree structure is shown in figure 5.7.

Function (RPN)					NC[0]
AND	AND	T	T	T	(1.3x2) x 1.3 +1.3 = 4.68
AND	OR	T	T	T	(1.2x2) x 1.3 +1.3 = 4.42
AND	NAND	T	T	T	(1.1x2) x 1.3 +1.3 = 4.16
AND	NOR	T	T	T	(1.2x2) x 1.3 +1.3 = 4.42
OR	AND	T	T	T	(1.3x2) x 1.2 +1.2 = 4.32
OR	OR	T	T	T	(1.2x2) x 1.2 +1.2 = 4.08
OR	NAND	T	T	T	(1.1x2) x 1.2 +1.2 = 3.84
OR	NOR	T	T	T	(1.2x2) x 1.2 +1.2 = 4.08
NAND	AND	T	T	T	(1.3x2) x 1.1 +1.1 = 3.96
NAND	OR	T	T	T	(1.2x2) x 1.1 +1.1 = 3.74
NAND	NAND	T	T	T	(1.1x2) x 1.1 +1.1 = 3.52
NAND	NOR	T	T	T	(1.2x2) x 1.1 +1.1 = 3.74
NOR	AND	T	T	T	(1.3x2) x 1.2 +1.2 = 4.32
NOR	OR	T	T	T	(1.2x2) x 1.2 +1.2 = 4.08
NOR	NAND	T	T	T	(1.1x2) x 1.2 +1.2 = 3.84
NOR	NOR	T	T	T	(1.2x2) x 1.2 +1.2 = 4.08

Table 5.3 - All Possible Root Node Values For Individuals Of Length 5

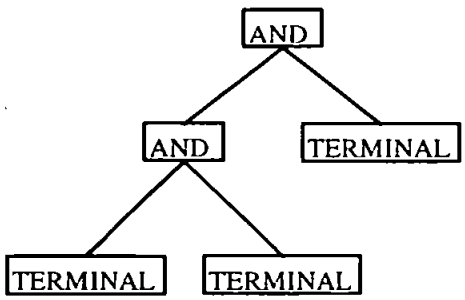


Figure 5.7 - Initial Structure Shown In Table 5.3

This produces 9 unique values from the lowest of 3.52 up to the highest of 4.68, with a range of 1.16, these are listed in table 5.4 together with their frequencies.

NC[0] value	frequency
4.68	1
4.42	2
4.32	1
4.16	1
4.08	4
3.96	1
3.84	2
3.74	2
3.52	1

Table 5.4 - All Unique Values Produced And Frequency Of Occurance

The run parameters are shown in table 5.5.

NC Max.	130.0
NC Min.	0.0
Elite	5
CCC	± 2.0 of NC value
Maximum Chromosome length	100
Max. generations	200

Table 5.5 - Run Parameters For The Parity 3 Problem

Fitness Calculation

The standardised fitness of an individual is the sum, over the 2^3 fitness cases, of the error between the value returned by the individual and the correct value of the particular Boolean function. Standardised fitness ranges between 0 and 2^3. The fitness calculation used (Koza 92, 94) involves the following calculation, where the fitness of each individual is the number of outputs of the problem, subtracted by the number of correct outputs of the individual being examined, i.e.

Fitness = Test points - hits.

This will return integer values for the fitness of the individual ranging from 0 to 8. This can lead to individuals with the same fitness values but vastly differing complexities. For example a solution with a fitness of 4.0 and a root node value of 8.88 should be ranked above another individual with the same fitness but a higher complexity. It was for this reason that the fitness measure was adjusted to:-

Fitness = (Test Points - Hits) + 0.001 x NC[0].

This then allows individuals of the same fitness but less complexity to be ranked above ones with higher complexity values. Due to the upper limit on the complexity of the individuals that can be produced, the root node complexity NC[0] will never exceed 1000.0 and so the additional fitness function term will never exceed 1.0 thus not affecting the hits criterion but allowing the population to be ranked with the least complex first.

Initial tests were performed and table 5.6 shows the preliminary results for the parity 3 problem. The computational effort for each run using DRAM-GP is based on 100 runs.

Method	Population size M (popsize x species)	IM	CM	Effort E
Koza,1992(STD)	4,000 (4,000x1)	n/a	n/a	80,000
Koza,1994(STD)	16,000 (16,000x1)	n/a	n/a	96,000
Koza,1994(ADF)	16,000 (16,000x1)	n/a	n/a	64,000
DRAM-GP	10 (10x1)	80	0.5	14,060
DRAM-GP	30 (30x1)	80	0.5	15,840
DRAM-GP	50 (50x1)	80	0.5	15,750
DRAM-GP	50 (10x5)	80	0.5	13,600
DRAM-GP	100 (10x10)	80	0.5	12,900
DRAM-GP	100 (20x5)	80	0.5	8,400
DRAM-GP	150 (10x15)	80	0.5	9,900
DRAM-GP	200 (10x20)	80	0.5	11,600
DRAM-GP	200 (20x10)	80	0.5	10,000
DRAM-GP	300 (20x15)	80	0.5	8,400
DRAM-GP	400 (20x20)	80	0.5	7,600

Table 5.6 - Initial Parity 3 Results

The initial results show an improvement over the conventional GP paradigm, however the results do not show which elements of the DRAM-GP algorithm are important for solving this problem. Further experiments are now undertaken to determine the effects of the injection mutation rate, the crossover mutation rate and the population size.

The first series of tests use a single population size of 10 individuals with crossover mutation rates of 0.0, 0.5 and 1.0. The injection mutation rate (see section 5.3.5) is initially turned off (I=0), and then set to 10, 30 and finally 50. This requires 12 sets of runs to cover all combinations of parameters. The results of these runs is shown in tables 5.7 to 5.10. Each row within the tables is calculated from 100 independent runs and this is repeated 10 times for each parameter set to produce an average value for the computational effort. A total of 1000 independent runs are produced for each parameter set, and the results averaged.

C=0.0 I=0 P=10x1		C=0.5 I=0 P=10x1		C=1.0 I=0 P=10x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
4%	18360	4%	45900	4%	112480
4%	22800	2%	50490	2%	119340
2%	27540	3%	59670	2%	146880
6%	30400	2%	59670	4%	160650
7%	34200	3%	123930	3%	264480
2%	34200	3%	129960	1%	422280
4%	41040	3%	252450	2%	453720
1%	146880	1%	270810	1%	633420
3%	161880	0%	-	1%	665550
1%	203450	0%	-	0%	-
3.3%	**72,075**	**2.1%**	**124,110***	**2.0%**	**330,978***

Table 5.7 - Even 3 Parity Problem, Population Size 10, I=0
(* average of available result)

C=0.0 I=10 P=10x1		C=0.5 I=10 P=10x1		C=1.0 I=10 P=10x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
5%	28,800	4%	27,540	3%	18,360
6%	38,760	5%	45,200	3%	43,320
5%	42,560	3%	47,880	7%	45,900
5%	44,100	4%	50,160	7%	47,250
5%	50,400	2%	50,490	2%	50,490
4%	52,440	3%	82,620	2%	105,570
6%	71,190	4%	84,360	3%	132,240
3%	87,210	4%	120,080	2%	133,110
6%	106,400	3%	136,800	1%	197,370
4%	136,730	4%	203,400	2%	312,360
4.9%	**65,859**	**3.6%**	**84,853**	**3.2%**	**108,597**

Table 5.8 - Even 3 Parity Problem, Population Size 10, I=10

C=0.0 I=30 P=10x1		C=0.5 I=30 P=10x1		C=1.0 I=30 P=10x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
5%	11400	3%	32120	3%	22950
4%	22950	4%	59280	2%	32130
8%	31360	5%	85120	7%	36480
3%	41310	2%	169830	5%	50490
6%	50160	2%	180120	5%	53110
4%	59280	2%	192780	4%	89680
5%	59280	2%	197370	4%	95760
3%	105570	3%	206550	2%	215730
1%	119340	2%	243270	1%	238680
2%	145920	1%	247860	2%	419520
4.1%	**64,657**	**2.6%**	**161,430**	**3.5%**	**125,453**

Table 5.9 - Even 3 Parity Problem, Population Size 10, I=30

C=0.0 I=50 P=10x1		C=0.5 I=50 P=10x1		C=1.0 I=50 P=10x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
4%	31,640	3%	18360	3%	13770
6%	36,000	5%	41310	2%	41310
6%	41,040	4%	57630	7%	51680
4%	45,600	1%	68850	4%	66120
5%	48,640	4%	73440	2%	78030
3%	79,040	3%	91800	3%	96390
4%	101,700	2%	109440	2%	100980
2%	123,930	3%	110160	1%	183600
3%	171,760	2%	152760	1%	449820
1%	348,840	1%	330480	0%	-
3.8%	**102,819**	**2.8%**	**105,423**	**2.5%**	**120,188***

Table 5.10 - Even 3 Parity Problem, Population Size 10, I=50

Even with a very small population size of only 10 individuals solutions to the problem are found. It must also be remembered that the top 5 individuals within each population are elite. The best results for this parameter set are obtained when the crossover mutation rate (see section 5.3.3), C, is set to 0.0 and the injection mutation rate set to 10 (the population size) with 65,859 individuals needing to be processed to produce a result with 99% certainty. Turning off the injection mutation operator has the effect of slightly increasing the computational effort to 72,075 individuals. An increase in the injection mutation operator further reduces performance. The results are used to produce the graph shown in figure 5.8.

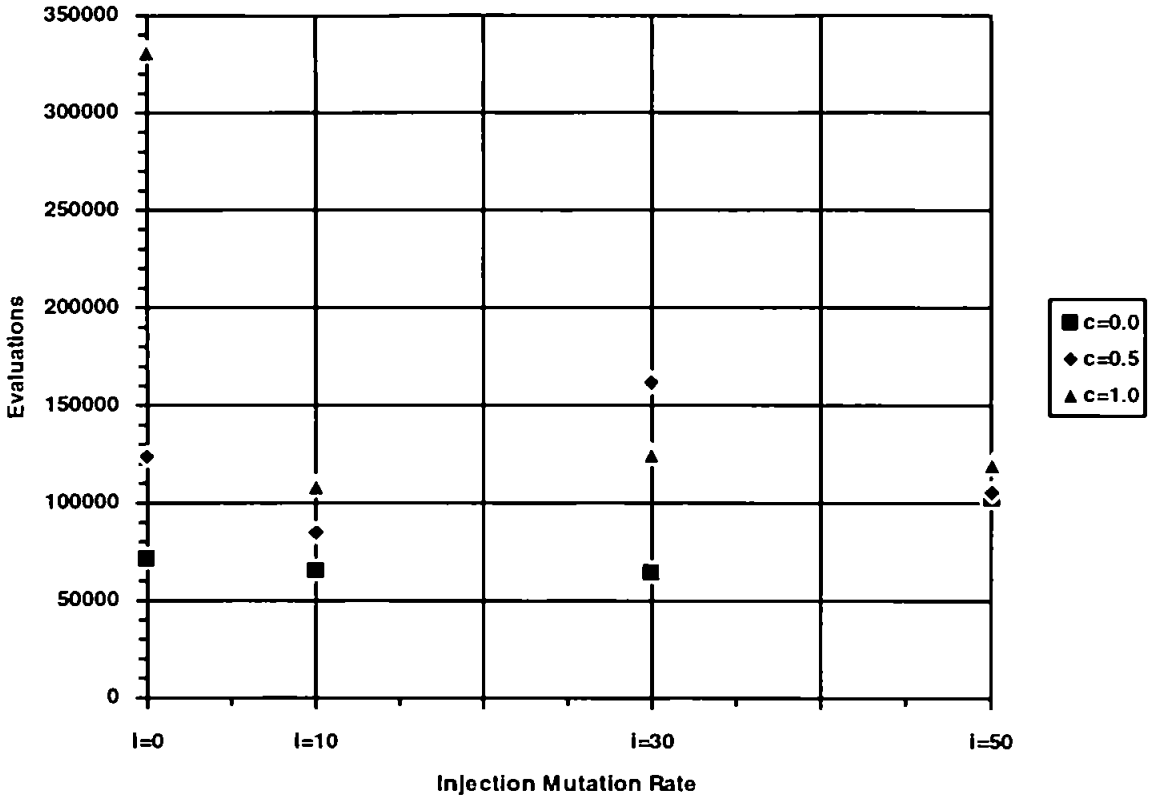


Figure 5.8 - Graph Of Evaluations Required To Solve The Even 3 Parity Problem With A Population Size Of 10 With Various Injection Mutation Rates.

The injection mutation rate will now be set to the population size based on these results. The population size is now increased to 20 individuals again all within one species group. The results are presented in table 5.11.

C=0.0 I=20 P=20x1		C=0.5 I=20 P=20x1		C=1.0 I=20 P=20x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
15%	15,000	10%	36,480	4%	68,400
15%	17,600	11%	45,000	8%	69,920
18%	24,200	8%	45,600	6%	73,440
15%	25,200	9%	48,640	8%	74,580
15%	27,360	9%	54,000	6%	76,000
14%	33,320	8%	75,580	10%	79,200
13%	36,480	6%	90,400	6%	95,400
12%	37,120	8%	97,180	9%	124,200
12%	40,500	6%	113,000	5%	164,160
14%	43,200	4%	155,040	3%	275,400
14.3%	**29,998**	**7.9%**	**76,092**	**6.5%**	**110,070**

Table 5.11 - Even 3 Parity Problem With A Population Size Of 20x1

Again it can be seen that the best results are achieved with no crossover mutation and requires only 29,998 individuals to be processed.

The next series of runs increases the population size to 30 individuals, one set of runs using only one species group (P=30x1), and another series of runs using 3 species of 10 individuals.

C=0.0 I=30 P=30x1		C=0.5 I=30 P=30x1		C=1.0 I=30 P=30x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
23%	17,640	12%	30,870	14%	34,200
23%	18,000	16%	32,400	12%	41,310
28%	19,530	11%	32,640	14%	51,450
32%	19,950	19%	35,640	13%	55,080
27%	21,420	12%	38,280	12%	67,800
24%	21,420	17%	46,200	6%	68,400
20%	23,100	18%	56,250	10%	88,140
22%	24,180	9%	60,750	11%	118,650
22%	24,300	9%	61,020	7%	124,200
13%	27,120	5%	123,120	7%	165,240
23.4%	**21,666**	**12.8%**	**51,717**	**10.6%**	**81,447**

Table 5.12 - Even 3 Parity Problem With A Population Size Of 30x1

C=0.0 I=30 P=10x3		C=0.5 I=30 P=10x3		C=1.0 I=30 P=10x3	
% runs	Effort E	% runs	Effort E	% runs	Effort E
59%	13,680	45%	9120	39%	27,000
61%	14,490	35%	13,500	44%	27,360
53%	15,840	49%	20,520	51%	29,640
62%	17,160	59%	21,420	51%	29,700
53%	17,640	54%	23,400	45%	32,130
55%	18,000	58%	23,700	38%	32,340
63%	18,870	49%	23,850	42%	33,600
59%	21,600	46%	29,580	42%	34,800
53%	24,480	47%	31,860	44%	41,520
51%	27,720	40%	37,440	40%	49,980
56.9%	**18,948**	**48.2%**	**23,439**	**43.6%**	**33,807**

Table 5.13 - Even 3 Parity Problem With A Population Size Of 10x3

Table 5.12 shows the best results for a single population run are achieved with no crossover mutation requiring 21,666 individuals to be processed. By splitting the population into sub-populations or species, shown in table 5.13, the number of individuals required to solve the problem is reduced to 18,948, again with no crossover mutation. Further results were obtained using larger population sizes, and are presented in tables 5.14 to 5.19.

C=0.0 I=40 P=40x1		C=0.5 I=40 P=40x1		C=1.0 I=40 P=40x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
40%	23760	25%	33600	7%	36,720
32%	26040	23%	34720	18%	60,760
32%	27200	18%	40960	14%	61,200
29%	28160	17%	47360	18%	66,640
33%	28800	18%	58080	11%	67,800
28%	29000	14%	60000	6%	79,040
29%	30160	18%	64000	8%	100,800
30%	34680	19%	72480	13%	101,920
16%	38000	13%	134750	11%	135,000
16%	84000	13%	148800	13%	180,800
28.5%	**34,980**	**17.8%**	**69,475**	**11.9%**	**89,068**

Table 5.14 - Even 3 Parity Problem With A Population Size Of 40x1

C=0.0 I=40 P=20x2		C=0.5 I=40 P=20x2		C=1.0 I=40 P=20x2	
78%	9,600	71%	16,120	66%	20,640
72%	11,880	64%	16,640	48%	21,280
72%	11,880	67%	19,200	59%	23,520
72%	12,960	60%	19,360	56%	25,200
79%	13,400	59%	22,680	52%	25,800
77%	13,920	59%	22,680	60%	26,800
74%	14,400	67%	23,400	52%	27,360
72%	14,880	67%	24,000	48%	30,240
70%	15,600	71%	24,000	51%	31,200
71%	17,480	61%	24,360	43%	32,240
73.7%	**13,600**	**64.6%**	**21,244**	**53.5%**	**26,428**

Table 5.15 - Even 3 Parity Problem With A Population Size Of 20x2

C=0.0 I=40 P=10x4		C=0.5 I=40 P=10x4		C=1.0 I=40 P=10x4	
% runs	Effort E	% runs	Effort E	% runs	Effort E
74%	15,360	69%	23400	64%	23,520
79%	16,120	63%	24480	50%	30,720
74%	16,560	62%	24600	55%	30,800
72%	16,640	68%	25760	54%	31,920
72%	18,240	61%	26400	54%	31,920
66%	20,400	60%	26600	48%	32,640
74%	20,440	62%	28160	58%	33,280
75%	20,800	58%	28800	62%	34,320
66%	22,080	52%	31000	53%	34,320
66%	25,920	58%	32000	56%	34,560
71.8%	**19,256**	**61.3%**	**27120**	**55.4%**	**31,800**

Table 5.16 - Even 3 Parity Problem With A Population Size Of 10x4

C=0.0 I=50 P=50x1		C=0.5 I=50 P=50x1		C=1.0 I=50 P=50x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
47%	17150	25%	34000	19%	35750
33%	20250	22%	37200	11%	58500
42%	20400	27%	37400	17%	61600
44%	21850	26%	43400	14%	70400
44%	25650	16%	44000	19%	72000
42%	26250	21%	48000	15%	73500
38%	27500	18%	48000	14%	91000
35%	29400	18%	66000	13%	107350
37%	29700	17%	77000	15%	120000
41%	33600	15%	77000	8%	330000
40.3%	**25,175**	**20.5%**	**51,200**	**14.5%**	**102,010**

Table 5.17 - Even 3 Parity Problem With A Population Size Of 50x1

C=0.0 I=80 P=20x4		C=0.5 I=80 P=20x4		C=1.0 I=80 P=20x4	
% runs	Effort E	% runs	Effort E	% runs	Effort E
94%	11,200	90%	15,040	86%	16,000
92%	11,200	88%	16,000	85%	18,720
95%	12,160	88%	16,240	90%	18,800
92%	14,080	86%	16,800	83%	20,640
97%	14,400	90%	17,280	84%	20,800
94%	14,400	87%	17,280	81%	21,120
92%	14,560	86%	18,560	82%	21,760
93%	14,720	87%	19,200	90%	22,320
95%	15,360	90%	19,520	83%	24,000
94%	15,680	89%	20,800	77%	24,960
93.8%	**13,776**	**88.1%**	**17,672**	**84.1%**	**20,912**

Table 5.18 - Even 3 Parity Problem With A Population Size Of 20x4

C=0.0 I=200 P=50x4		C=0.5 I=200 P=50x4		C=1.0 I=200 P=50x4	
% runs	Effort E	% runs	Effort E	% runs	Effort E
100%	11400	87%	32000	93%	21,600
100%	12000	77%	32000	99%	22,200
100%	12000	78%	32200	100%	23,200
98%	12000	81%	32400	99%	23,400
100%	12600	85%	32400	95%	24,600
100%	13200	84%	33600	97%	25,000
100%	13600	76%	35200	97%	25,200
99%	13600	84%	36400	99%	26,400
100%	13800	82%	43200	97%	26,400
99%	14000	79%	44600	91%	27,000
99.6%	**12,820**	**81.3%**	**35,400**	**96.7%**	**24,500**

Table 5.19 - Even 3 Parity Problem With A Population Size Of 50x4

Figure 5.9 shows the results of using only single populations for solving the Even 3 Parity problem.

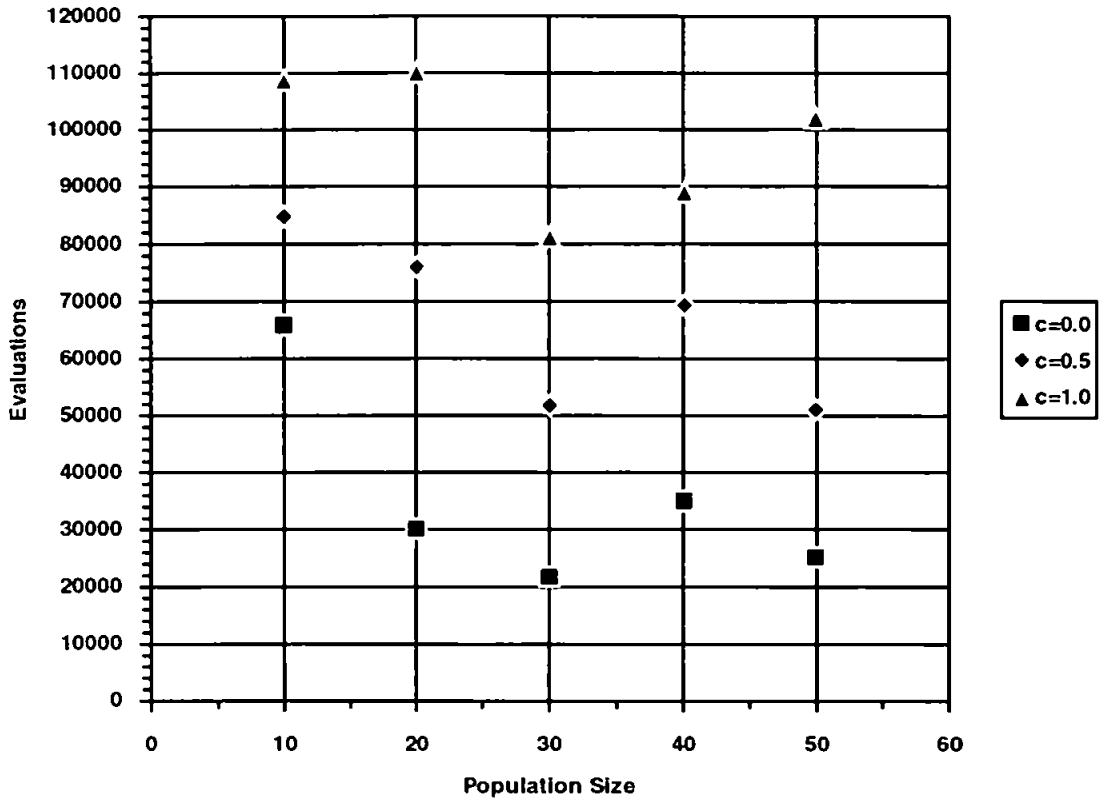


Figure 5.9- Single Population Even 3 Parity Results

Figure 5.9 clearly shows that the crossover mutation rate has a direct bearing on the computational expense required to solve the problem. The better results for each population size are achieved with no crossover mutation. The best overall result is produced using a population size of 30, and required 21,666 individuals to be processed. The worst results occur with a crossover mutation rate of 1.0. This seems to confirm the idea that fit sub-trees are being crossed, improving the fitness of successive individuals by not disrupting the swapped sub-trees. If the best results were obtained using a crossover mutation rate of 1.0 then the disruption to the sub-tree would be too large and the idea of useful sub-trees contributing to higher fitness individuals would be invalid.

Figure 5.10 shows the effect of the crossover mutation rate on the multi-population runs.

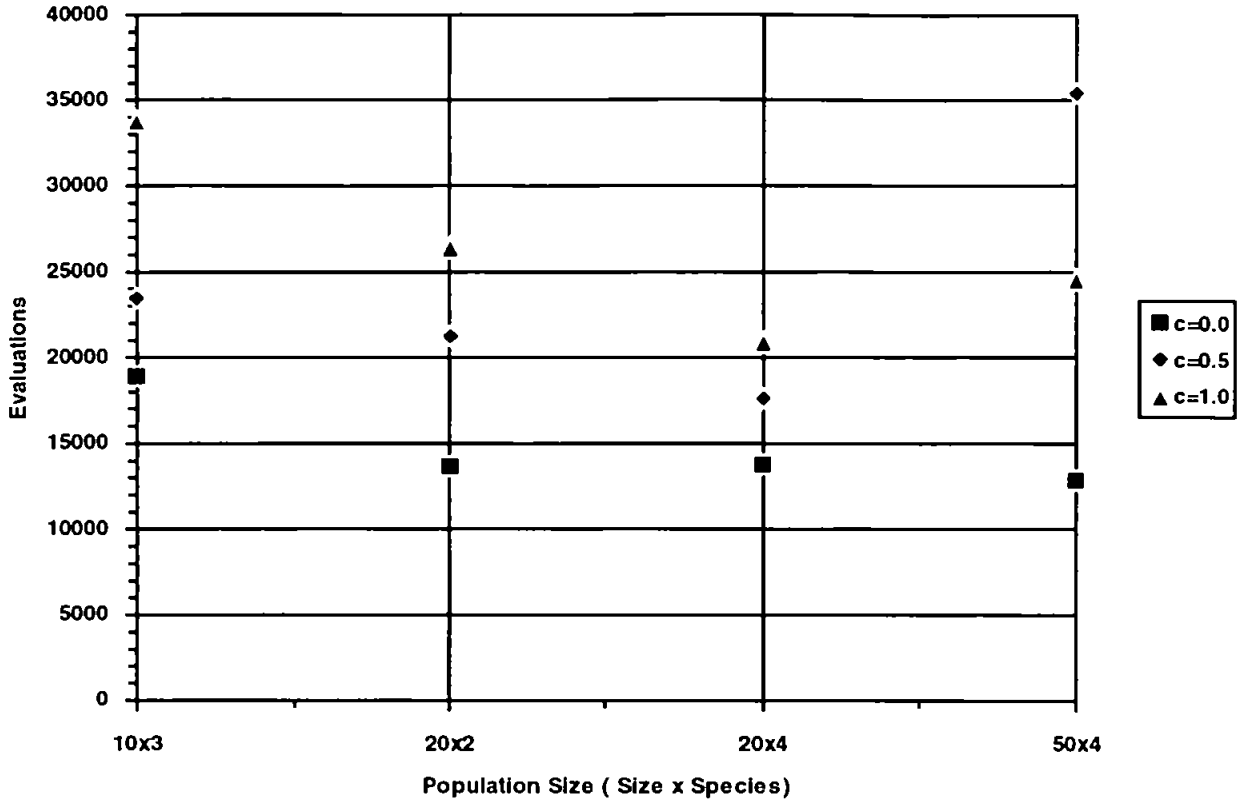


Figure 5.10- Multi-Population Even 3 Parity Results

As with the previous results the best results for the multi-populations are achieved with no crossover mutation but the number of individuals that need to be processed is reduced, the lowest being 12,820 using 4 species each consisting of 50 individuals. The results of the 20x2 and 20x4 populations are also low, both being below 15,000 evaluations. So with a population size 200 times smaller than the results published in (Koza, 1992) a result is produced using 6.24 times less evaluations.

5.6.2 The Even 4 Parity Problem

This problem is as parity 3 but using 4 input variables (Koza, 92, 94). The goal function of the even 4 parity function *f* of 4 variables D0, D1, D2 and D3 is shown in table 5.20.

no.	D3	D2	D1	D0	Output *f*
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	1

Table 5.20 - Truth Table For Even 4 Parity Problem

The run parameters are the same as the parity 3 parameters except the terminal set is increased to T={d0,d1,d2,d3}, the maximum chromosome length is 256, test points=16, maximum node complxity=320, and minimum node complexity=200. The crossover mutation rate was set to 0.5 for all runs. Table 5.21 shows a comparison of standard GP and DRAM-GP.

Method	Population size M (popsize x species)	Effort E
Koza,1992(STD)	4000 (4000x1)	1,276,000
Koza,1994(STD)	16000 (16000x1)	384,000
Koza,1992(ADF)	4000 (4000x1)	88,000
Koza,1994(ADF)	16000 (16000x1)	176,000
DRAM-GP	30 (30x1)	198,450
DRAM-GP	50 (50x1)	215,600
DRAM-GP	50 (10x5)	195,300
DRAM-GP	80 (80x1)	169,520
DRAM-GP	100 (100x1)	144,000
DRAM-GP	100 (10x10)	267,900
DRAM-GP	100 (20x5)	102,600
DRAM-GP	120 (120x1)	192,000
DRAM-GP	150 (10x15)	145,800
DRAM-GP	150 (30x5)	111,600
DRAM-GP	200 (10x20)	145,200
DRAM-GP	200 (20x10)	104,400
DRAM-GP	200 (40x5)	95,400
DRAM-GP	250 (50x5)	87,000
DRAM-GP	300 (20x15)	47,700
DRAM-GP	300 (30x10)	83,400
DRAM-GP	400 (40x10)	56,400
DRAM-GP	500 (50x10)	78,500

Table 5.21 - Even 4 Parity Results

Table 5.21 shows that using a population size of 300, consisting of 10 species of 30 individuals, a total of 47,700 individuals need to be processed to provide a solution with a 99% probability of success. As with the 3 parity results the population size required to solve this problem is greatly reduced with an increase in performance.

5.6.3 The Even 5 Parity Problem

This problem has 5 input variables (Koza, 92, 94). Run parameters are as parity 3 except that the terminal set is increased to T={d0,d1,d2,d3,d4}, the maximum chromosome length is 600, number of test points=32, maximum node complexity=600, and minimum node complexity=430. Table 5.22 shows a comparison of standard GP with DRAM-GP.

Method	Population size M (popsize x species)	Effort E
Koza, 1992 (STD)	16,000 (16,000x1)	6,528,000
Koza, 1994 (ADF)	4,000 (4,000x1)	152,000
Koza, 1994 (ADF)	16,000 (16,000x1)	464,000
DRAM-GP	80 (80x1)	2,119,680
DRAM-GP	250 (25x10)	5,128,250
DRAM-GP	250 (50x5)	5,137,000
DRAM-GP	500 (500x1)	2,048,500
DRAM-GP	500 (100x5)	1,260,000
DRAM-GP	500 (50x10)	3,870,000

Table 5.22 - Even 5 Parity Results

The results presented in table 5.22 show an improvement over standard GP but perform less well against automatically defined functions (ADF's) (Koza, 1994). The computational effort required to solve the problem is about ten times greater than GP using ADF's but the population size is eight times smaller. ADF's are well suited to the parity class of problems, if a sub-function is produced which solves the even 3 parity problem then 25% of the fitness cases are correct. The solution for the even 5 parity problem can be described using the even 3 parity sub-tree (Koza, 19924) which again confirms the suitability of this problem when using ADF's.

5.6.4 The 6-Multiplexer Problem

The input to the Boolean N-multiplexer function is the Boolean value (0 or 1) of the particular data bit that is singled out by the k address bits a_i and 2^k data bits d_i, where $N=k+2^k$. The experiments presented here have $k=2$, i.e. the 6-multiplexer. The DRAM-GP parameters are shown in table 5.23.

Functional set	F ={and ,or ,not ,if }
Arguments	F_A={ 2, 2, 1, 3 }
NC Functionals	N_F={1.2, 1.2, 1.1,1.3 }
Terminal set	T={a0,a1,d0,d1,d2,d3}
NC Terminals	N_T={1,1,1,1,1,1}
Crossover Mutation Rate	0.5
Imutation (IM)	every 80 evaluations
Test points (TP)	64
Fitness	(TP-Hits)+0.001NC[0]
Max. NC	60.0
Min. NC	5.0
Elite	5
CCC	± 2.0 of NC value
Chromosome length	100
Max. generations	200

Table 5.23 - Run Parameters For 6-Multiplexer Problem

In order to maintain the high level of mutation using DRAM-GP sub-tree repair is required. If a NOT function (arity 1) is mutated into a OR/AND (arity 2) function, an extra argument is required and so a terminal is randomly chosen from the set T to ensure a correct function, this process is repeated if a NOT is mutated into an IF function. End sub-trees are deleted when mutating from IF to AND/OR, IF to NOT, and AND/OR to NOT. This process ensures that the mutation can be unbiased in selecting new functionals, and also ensures that all mutated individuals are closed and valid tree structures. After initial runs the best performance was achieved using the node complexity values presented in table 5.23. The values are similar in magnitude to the values used for the even parity problems presented earlier. Any change in node complexity values disrupted the CCC mechanism and so values

for the node complexity are set to unity for terminals and slightly larger than unity for functionals.

Table 5.24 shows a comparison of GP with DRAM-GP for the 6-multiplexer.

Method	Population size	Effort E
Koza, 94	500	245,000
Koza, 94	1000	343,000
Koza, 94	2000	200,000
Koza, 94	4000	160,000
Koza, 94 greedy over selection	1000	33,000
Koza, 94 greedy over selection	2000	18,000
Koza, 94 greedy over selection	4000	24,000
Koza, 94 tournament selection	1000	123,000
DRAM-GP	10 (10x1)	491,130
DRAM-GP	20 (10x2)	15,600
DRAM-GP	30 (30x1)	300,960
DRAM-GP	40 (20x2)	13,320
DRAM-GP	50 (50x1)	109,950
DRAM-GP	50 (10x5)	14,250
DRAM-GP	80 (80x1)	65,520
DRAM-GP	100 (100x1)	90,000
DRAM-GP	100 (10x10)	16,500
DRAM-GP	100 (20x5)	14,100

Table 5.24 - Initial 6-Multiplexer Results

As with the parity problems, further runs were produced to determine the effect of the various parameters and are presented in tables 5.25 to 5.33. The parameters changed are the crossover mutation rate, C, and the population size and number of species. The injection mutation rate is usually set to the total population size. All results for computational effort (Koza,1992) are calculated by producing 100 runs and repeating this 10 times, the average of the 10 runs is then calculated, producing 1000 runs for each parameter set.

C=0.0 I=10 P=10x1		C=0.5 I=10 P=10x1		C=1.0 I=10 P=10x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
2%	111720	7%	70200	2%	64,260
5%	104880	2%	173280	5%	85,120
5%	82620	4%	66880	4%	91,800
8%	59280	8%	36720	4%	95,760
5%	105570	4%	170630	4%	103,360
5%	45900	2%	201960	5%	120,840
2%	87210	9%	71440	3%	133,110
8%	70680	3%	284240	1%	137,700
2%	169830	7%	75240	3%	279,680
6%	62100	2%	119340	1%	362,610
4.8%	**89,979**	**4.8%**	**126,993**	**3.2%**	**147,424**

Table 5.25 - 6-Multiplexer Problem With A Population Size Of 10

C=0.0 I=20 P=10x2		C=0.5 I=20 P=10x2		C=1.0 I=20 P=10x2	
% runs	Effort E	% runs	Effort E	% runs	Effort E
49%	18,960	54%	15,840	56%	15,600
59%	18,180	51%	13,600	65%	15,860
50%	20,400	51%	19,580	57%	16,320
54%	16,380	57%	12,960	59%	17,280
47%	14,960	53%	15,840	52%	17,400
56%	16,800	56%	16,800	49%	17,760
47%	19,400	41%	24,240	48%	18,360
53%	14,760	52%	14,400	53%	18,720
47%	19,880	55%	12,400	53%	20,400
46%	19,500	50%	19,740	41%	25,840
50.8%	**17,922**	**52.0%**	**16,540**	**53.3%**	**18,354**

Table 5.26 - 6-Multiplexer Problem With A Population Size Of 20 (10x2)

C=0.0 I=20 P=20x1		C=0.5 I=20 P=20x1		C=1.0 I=20 P=20x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
14%	83,080	26%	48,000	24%	31,960
22%	61,600	25%	63,580	24%	39,680
17%	74,500	18%	87,840	15%	46,640
18%	75,000	16%	82,080	20%	47,040
22%	50,800	18%	87,500	18%	50,160
27%	55,500	19%	77,880	16%	53,700
14%	114,700	17%	93,500	16%	54,560
16%	106,920	15%	104,400	17%	54,880
23%	69,160	18%	90,000	16%	56,840
17%	40,500	21%	78,540	11%	67,100
19.0%	**73,176**	**19.3%**	**81,332**	**17.7%**	**50,256**

Table 5.27 - 6-Multiplexer Problem With A Population Size Of 20 (20x1)

C=1.0 I=30 P=30x1		C=0.0 I=30 P=10x3		C=0.5 I=30 P=10x3		C=1.0 I=30 P=10x3	
% runs	Effort E	% runs	Effort E	% runs	Effort E	% runs	Effort E
46%	24,480	60%	35,280	60%	35,460	70%	18,000
36%	29,070	54%	34,020	61%	33,660	71%	20,100
36%	31,050	65%	26,700	62%	29,850	64%	21,360
33%	34,020	67%	27,600	59%	35,460	67%	21,840
33%	36,400	56%	32,040	63%	27,750	65%	23,400
29%	37,200	59%	32,220	59%	34,020	63%	24,030
32%	38,940	58%	35,100	60%	34,740	63%	24,600
26%	41,820	58%	33,120	68%	30,000	59%	24,960
23%	42,750	55%	35,280	66%	30,000	63%	27,720
25%	62,160	58%	34,200	68%	26,250	52%	28,080
31.9%	**37,789**	**59.0%**	**32,556**	**62.6%**	**31,719**	**63.7%**	**23,409**

Table 5.28 - 6-Multiplexer Problem With A Population Size Of 30

C=1.0 I=40 P=40x1		C=1.0 I=40 P=10x4	
% runs	Effort E	% runs	Effort E
50%	26,800	83%	15,800
50%	30,240	83%	16,320
45%	32,400	84%	16,560
48%	32,640	78%	16,640
45%	32,640	80%	17,200
44%	35,840	78%	18,400
43%	33,440	82%	18,600
43%	36,480	75%	18,960
43%	36,480	83%	19,360
44%	36,720	79%	21,200
45.5%	**33,368**	**80.5%**	**17,904**

Table 5.29 - 6-Multiplexer Problem With A Population Size Of 40

C=1.0 I=50 P=50x1		C=1.0 I=50 P=10x5	
% runs	Effort E	% runs	Effort E
65%	22,400	90%	11,700
66%	25,200	92%	13,000
62%	25,300	89%	15,600
61%	26,000	88%	15,600
55%	28,000	89%	16,250
59%	30,000	93%	16,950
61%	31,500	91%	17,500
52%	34,450	89%	17,600
48%	35,000	90%	17,600
51%	43,200	91%	18,500
58.0%	**30,105**	**90.2%**	**16,030**

Table 5.30 - 6-Multiplexer Problem With A Population Size Of 50

C=1.0 I=80 P=80x1		C=0.0 I=80 P=20x4		C=1.0 I=80 P=20x4	
% runs	Effort E	% runs	Effort E	% runs	Effort E
80%	21,600	98%	12,720	99%	10,240
78%	23,040	98%	13,200	100%	10,880
75%	24,960	97%	10,240	96%	11,200
77%	25,760	92%	16,800	98%	11,520
76%	25,760	97%	14,880	98%	13,200
70%	25,920	99%	10,080	96%	13,000
75%	27,360	99%	13,200	99%	13,120
71%	30,080	97%	13,120	100%	13,600
70%	30,720	98%	16,320	99%	14,400
72%	36,480	96%	13,600	95%	16,320
74.4%	**27,168**	**97.1%**	**13,416**	**98.0%**	**12,748**

Table 5.31 - 6-Multiplexer problem with a population size of 80

C=1.0 I=0 P=20x4	
% runs	Effort E
97%	16,320
87%	19,200
91%	20,480
96%	20,880
94%	21,120
93.0%	**19,600**

Table 5.32 - 6-Multiplexer With A Population Size Of 80 And No Injection Mutation

C=1.0 I=100 P=100x1	
% runs	Effort E
85%	20,500
85%	24,000
86%	24,400
80%	26,800
83%	28,200
81%	28,200
82%	28,800
84%	30,000
76%	30,100
76%	37,800
81.8%	**27,880**

Table 5.33 - 6-Multiplexer problem with a population size of 100

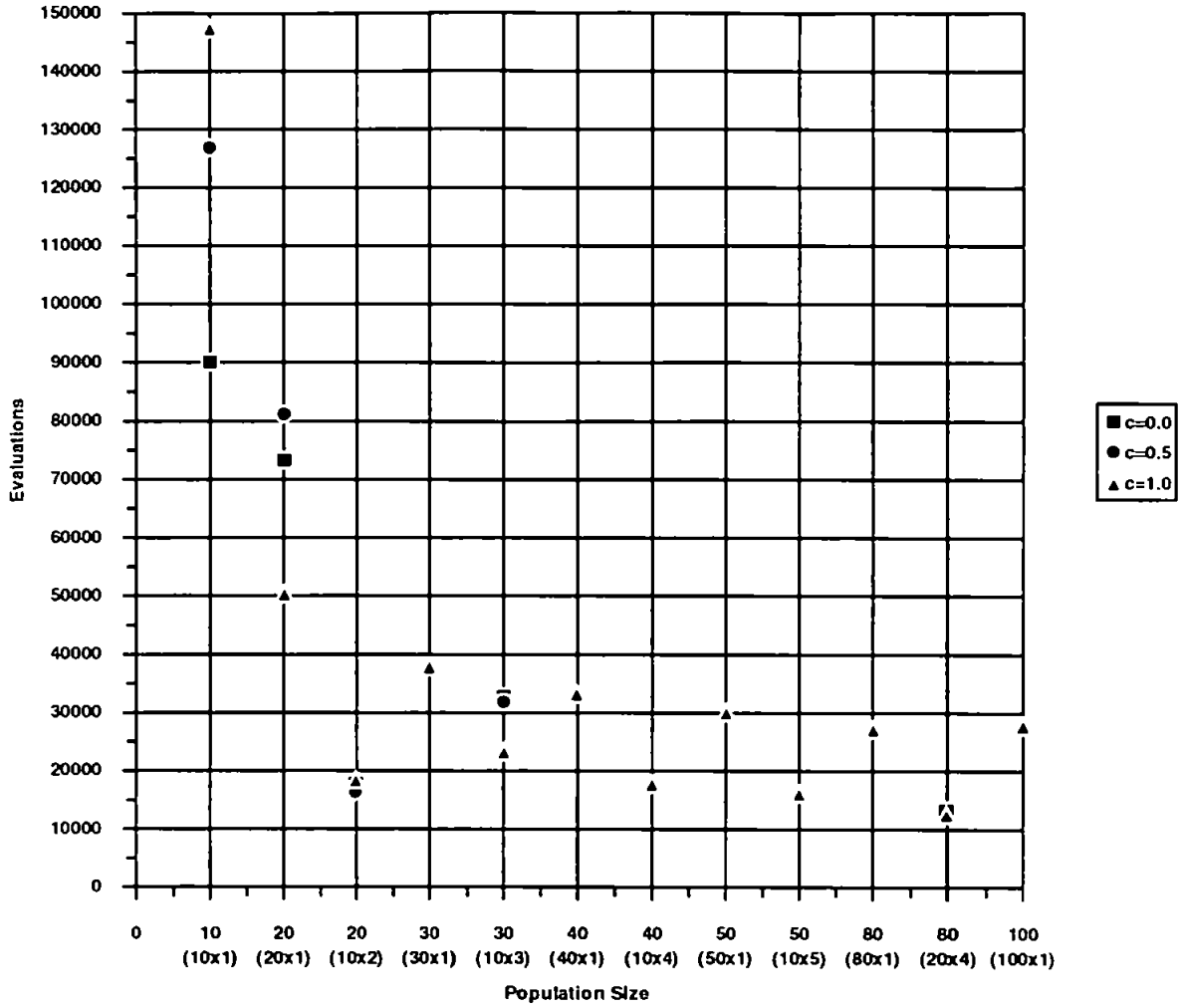


Figure 5.11- 6-Multiplexer Results

Figure 5.11 shows the results from tables 5.25 to 5.33. The results clearly show that in every case the multi-populations performed better than the single population runs. The best result is achieved using a population size of 80 (4 species of 20 individuals) requiring a total of 12,748 evaluation to obtain a solution with a probability of success of 99%. It is also interesting to note that there is only a small improvement in the performance when the population size is increased above 40 individuals. The variation of performance produced by the crossover mutation rate for this problem shows little effect except for the very small population sizes. The results are comparable with GP using ADF's (Koza, 1994) but requiring only 80 individuals compared with 2000 with ADF's.

5.7 Symbolic Regression

It has been shown that the new method produces very good results for Boolean induction problems as shown in section 5.6. Systems identification problems, tested earlier in the thesis, will now be tested using the improved GP paradigm.

5.7.1 The Two-Box Problem

The two-box problem concerns the identification of a relationship between six independent variables (x_1 , ... , x_6), where this relationship relates to the difference y in the volumes of the first box whose length, width, and height are x_1 , x_2 , x_3 and the second box whose length, width, and height are x_4 , x_5 , x_6 . Thus:- $y = (x_1\ x_2\ x_3) - (x_4\ x_5\ x_6)$. The goal of this symbolic regression is to derive the above equation as a "complete form" when given a set of N observations.

In this problem, where the raw fitness is a floating point number rather than an integer, there is no need to include the NC[0] weighting in the fitness calculation. The fitness is calculated using the mean squared error (MSE) of all of the test points.

The multiplication and divide functions are considered more complex than the plus and minus functions and thus have higher N_F values. Using 10 sets of 6 data points ranging from 0.0-10.0, and a functional set of: F = { +, -, *, % },

with N_F values of: N_F = {1.1, 1.1, 1.2, 1.2} respectively.

The terminal set is: T = { a1, a2, a3, a4, a5, a6 }

where a1,a2 and a3 are the length, width and height of box 1 and a4,a5 and a6 are the length, width and height of box 2.

Using various tree generation methods, and various tree sizes, random trees can be produced to attempt to solve the two-box problem. The results of these are presented in table 5.34.

creation method	individuals produced	max layer	max length	min length	avg length	best fit (MSE)	worst fit (MSE)	average fit (MSE)
grow	1,000,000	1	7	3	4.6015	3722.356	2.1213 E7	5.3622 E4
grow	1,000,000	2	15	3	5.8725	2018.921	8.0341 E13	1.2371 E7
grow	1,000,000	3	31	3	6.9075	2018.921	5.7103 E16	2.7581 E11
grow	1,000,000	4	49	3	7.7212	2573.798	3.6681 E20	3.7083 E14
grow	1,000,000	5	65	3	8.3705	2269.796	7.5757 E20	7.8406 E14
grow	1,000,000	6	83	3	8.8938	2556.219	1.7160 E22	2.8181 E16
full	1,000,000	1	3	3	3.0000	38526.410	5.2948 E4	4.1558 E4
full	1,000,000	2	7	7	7.0000	3722.357	2.1214 E7	9.8201 E4
full	1,000,000	3	15	15	15.0000	1846.201	1.0518 E14	1.6482 E9
full	1,000,000	4	31	31	31.0000	1976.505	1.7782 E21	2.4647 E15
full	1,000,000	5	63	63	63.0000	1469.860	1.8023 E30	1.8324 E24
full	1,000,000	6	127	127	127.0000	2447.048	∞	∞

Table 5.34 - Average Fitnes Of Various Tree Representations For The Two-Box Problem

Table 5.34 shows that even after 12,000,000 random individuals have been processed, no solutions are found.

The DRAM-GP parameters used for the two-box problem are shown in table 5.35.

Functional set	F ={ + , - , * , % }
Arguments	F_A ={ 2, 2, 2, 2 }
NC Functionals	N_F={1.1,1.1,1.2,1.2}
Terminal set	T={x1,x2,x3,x4,x5,x6}
NC Terminals	N_T={1,1,1,1,1,1}
Test points (TP)	10
Fitness	MSE
Elite	5
CCC	± 2.0 of NC value unless stated
Max. Chromosome length	50
Max. generations	200

Table 5.35 - Run Parameters For The Two-Box Problem

Table 5.36 shows published results (Koza, 1992, 994). The important point to note is the population size required to solve the problem, at 4000 individuals, the memory required to initiate a run is excessive and the high population size is reflected in the amount of processing required to produce a solution.

	Population size M	Effort E
Koza, 1992 (STD)	4000	1,176,000
Koza, 1994 (ADF)	4000	2,220,000

Table 5.36 - Standard GP Twobox Problem Results

As with previous tests, the population size and the crossover mutation rates are set at various values to assess the affects on the amount of processing required.

C=0.0 I=10 P=10x1		C=0.5 I=10 P=10x1		C=1.0 I=10 P=10x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
4%	134,470	22%	31,820	13%	40,670
5%	141,300	17%	32,550	12%	42,800
5%	147,600	13%	34,300	12%	43,200
4%	150,480	13%	36,750	13%	45,900
5%	155,700	14%	41,440	10%	48,400
4%	200,010	9%	52,480	9%	48,950
3%	250,800	12%	55,550	9%	63,750
3%	255,360	9%	58,850	7%	67,500
3%	259,930	9%	79,010	7%	69,300
3%	304,000	9%	95,760	7%	76,500
3.9%	**199,965**	**12.7%**	**51,851**	**9.9%**	**54,697**

Table 5.37 - Two-Box Problem With A Population Size of 10

C=0.0 I=20 P=10x1		C=0.5 I=20 P=10x1		C=1.0 I=20 P=10x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
3%	221,920	13%	41,800	12%	38,280
1%	472,770	11%	51,750	13%	44,000
1%	633,420	10%	53,100	12%	50,160
1%	711,450	12%	54,450	11%	54,400
1%	729,810	7%	58,760	8%	56,320
0%	∞	10%	59,840	7%	67,800
0%	∞	6%	63,900	6%	91,200
0%	∞	8%	72,960	4%	120,840
0%	∞	8%	73,150	6%	122,250
0%	∞	8%	91,300	5%	127,800
0.7%	**276,937***	**9.3%**	**62,101**	**8.4%**	**77,305**

(* indicates average of available results)

I=10 P=10x1 CCC=5.0		
C	% runs	Effort E
0.0	5%	201,140
0.0	2%	355,680
0.5	5%	91,800
0.5	5%	146,900
0.5	5%	167,400
1.0	10%	48,950
1.0	9%	62,720

Table 5.38 - Two-Box Problem With A Population Size of 10 and Constrained Complexity Crossover Of ± 5.0

C= 0.5 P=10x1		
I	% runs	Effort E
30	11%	65,600
40	6%	87,300
50	10%	79,100
60	13%	43,600
70	7%	48,590
80	6%	68,400
90	8%	88,920
100	14%	58,000

Table 5.39 - Two-Box Problem With A Population Size Of 10 And Various Injection Mutation Values

C=0.0 I=20 P=10x2		C=0.5 I=20 P=10x2		C=1.0 I=20 P=10x2		C=1.0 P=20x1 I=30	
% runs	Effort E	% runs	Effort E	% runs	Effort E	% runs	Effort E
10%	103,360	17%	53,040	17%	52,780	12%	80,360
8%	133,200	15%	71,540	16%	69,520	11%	82,500
6%	153,000	12%	81,340	12%	70,400	12%	92,400
5%	156,060	12%	88,200	11%	78,320	10%	94,600
8%	181,800	14%	96,720	12%	90,200	9%	97,180
5%	191,520	11%	99,000	9%	121,600	9%	128,380
5%	221,480	11%	100,100	10%	124,200	8%	140,400
5%	257,400	8%	112,500	10%	128,000	9%	149,600
5%	270,560	8%	136,960	8%	132,000	8%	169,500
5%	288,800	7%	183,040	5%	243,200	6%	328,320
6.2%	**195,718**	**11.5%**	**102,244**	**11.0%**	**111,022**	**9.4%**	**136,324**

Table 5.40 - Two-Box Problem With A Population Size Of 20

C=0.0 I=30 P=30x1		C=0.5 I=30 P=30x1		C=1.0 I=30 P=30x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
32%	41,400	43%	32,340	41%	31,620
29%	46,170	31%	38,760	40%	34,200
33%	46,410	34%	39,330	30%	40,500
33%	48,960	38%	42,750	33%	42,300
34%	49,140	33%	43,200	36%	42,840
27%	56,070	35%	45,000	35%	43,680
28%	63,000	32%	46,620	36%	45,000
27%	66,660	33%	48,840	31%	50,430
24%	69,600	36%	51,300	33%	56,760
24%	73,530	22%	87,120	29%	59,220
34.4%	**56,094**	**33.7%**	**47,526**	**34.4%**	**44,655**

Table 5.41 - Two-Box Problem With A Population Size Of 30x1

C=0.0 I=30 P=10x3		C=0.5 I=30 P=10x3		C=1.0 I=30 P=10x3	
% runs	Effort E	% runs	Effort E	% runs	Effort E
12%	117,600	14%	73,260	20%	74,370
11%	120,960	17%	80,190	17%	76,440
15%	122,400	19%	81,030	18%	88,740
8%	136,320	14%	86,700	15%	91,200
8%	144,000	18%	93,930	16%	98,790
9%	201,390	19%	98,010	12%	108,780
10%	203,520	7%	117,000	16%	115,200
8%	220,800	16%	120,930	15%	115,200
7%	303,750	14%	120,990	8%	140,400
6%	384,750	11%	145,200	11%	169,950
9.4%	**195,549**	**14.9%**	**101,724**	**14.8%**	**107,907**

Table 5.42 - Two-Box Problem With A Population Size Of 10x3

C=0.0 I=40 P=10x4		C=0.5 I=40 P=10x4		C=1.0 I=40 P=10x4	
% runs	Effort E	% runs	Effort E	% runs	Effort E
22%	72,080	38%	41,600	31%	63,240
20%	96,120	27%	62,560	28%	69,120
15%	113,600	25%	64,960	25%	71,000
15%	114,400	24%	80,000	26%	83,160
14%	138,240	24%	85,560	24%	84,000
14%	153,120	26%	87,000	22%	84,000
13%	153,680	19%	100,440	22%	93,240
14%	166,400	19%	103,880	17%	94,720
13%	174,000	20%	118,320	22%	98,280
14%	202,240	19%	124,000	20%	107,800
15.4%	**138,388**	**24.1%**	**86,832**	**23.7%**	**84,856**

Table 5.43 - Two-Box Problem With A Population Size Of 10x4

C=0.0 I=40 P=20x2		C=0.5 I=40 P=20x2		C=1.0 I=40 P=20x2	
% runs	Effort E	% runs	Effort E	% runs	Effort E
25%	72,800	38%	46,200	33%	47,120
22%	99,960	29%	58,320	32%	53,280
19%	103,000	28%	64,800	28%	57,240
21%	104,720	23%	68,000	24%	62,640
21%	109,040	27%	72,000	26%	63,840
18%	119,560	26%	79,360	31%	64,200
15%	121,600	24%	80,000	31%	67,320
19%	125,280	26%	81,600	28%	68,800
13%	143,360	26%	81,600	28%	68,800
14%	215,040	20%	90,280	20%	84,680
18.7%	**121,436**	**26.7%**	**72,216**	**28.1%**	**63,792**

Table 5.44 - Two-Box Problem With A Population Size Of 20x2

C=0.0 I=50 P=50x1		C=0.5 I=50 P=50x1		C=1.0 I=50 P=50x1	
% runs	Effort E	% runs	Effort E	% runs	Effort E
45%	52,500	48%	42,000	54%	37,600
37%	63,700	52%	43,500	54%	39,600
38%	65,000	44%	46,800	53%	40,500
40%	65,650	43%	48,000	52%	44,000
36%	67,150	45%	50,500	52%	45,500
31%	69,750	41%	51,000	52%	45,500
36%	76,500	43%	55,250	47%	46,350
30%	80,100	41%	60,600	50%	46,800
33%	86,700	43%	62,700	44%	47,500
29%	88,000	37%	68,250	40%	58,500
35.5%	**71,505**	**43.7%**	**52,860**	**49.8%**	**45,185**

Table 5.45 - Two-Box Problem With A Population Size Of 50x1

C=1.0 I=50 P=25x2		C=1.0 I=50 P=10x5	
% runs	Effort E	% runs	Effort E
39%	46,800	28%	62,350
35%	53,550	32%	66,000
37%	57,950	31%	71,400
36%	60,000	30%	78,850
31%	60,300	25%	87,000
37%	60,350	27%	87,500
33%	61,000	29%	93,600
38%	62,050	28%	97,500
37%	63,000	26%	98,600
28%	87,500	25%	107,300
35.1%	**61,250**	**28.1%**	**85,010**

Table 5.46 - Two-Box Problem With A Population Size Of 25x2 And 10x5

C=1.0 I=80 P=80x1		C=1.0 I=80 P=40x2		C=1.0 I=80 P=20x4		C=1.0 I=80 P=10x8	
% runs	Effort E	% runs	Effort E	% runs	Effort E	% runs	Effort E
67%	45,360	55%	49,280	62%	43,670	32%	81,600
65%	48,800	50%	50,400	51%	47,520	35%	85,680
57%	49,280	47%	60,480	49%	50,400	33%	94,800
59%	53,760	41%	61,360	50%	58,080	37%	96,000
52%	53,760	45%	68,640	45%	63,200	33%	100,320
57%	54,400	48%	69,600	43%	70,080	32%	100,640
57%	57,600	45%	70,560	42%	72,800	36%	102,000
60%	58,800	42%	72,080	40%	73,200	32%	104,960
58%	61,600	34%	88,400	38%	77,280	34%	116,160
53%	68,400	31%	106,400	49%	82,160	28%	124,000
58.5%	**55,176**	**43.8%**	**69,720**	**46.9%**	**63,839**	**33.2%**	**100,616**

Table 5.47 - Two-Box Problem With A Population Size Of 80

Tables 5.37 to 5.47 are used to produce the graph shown in figure 5.12.

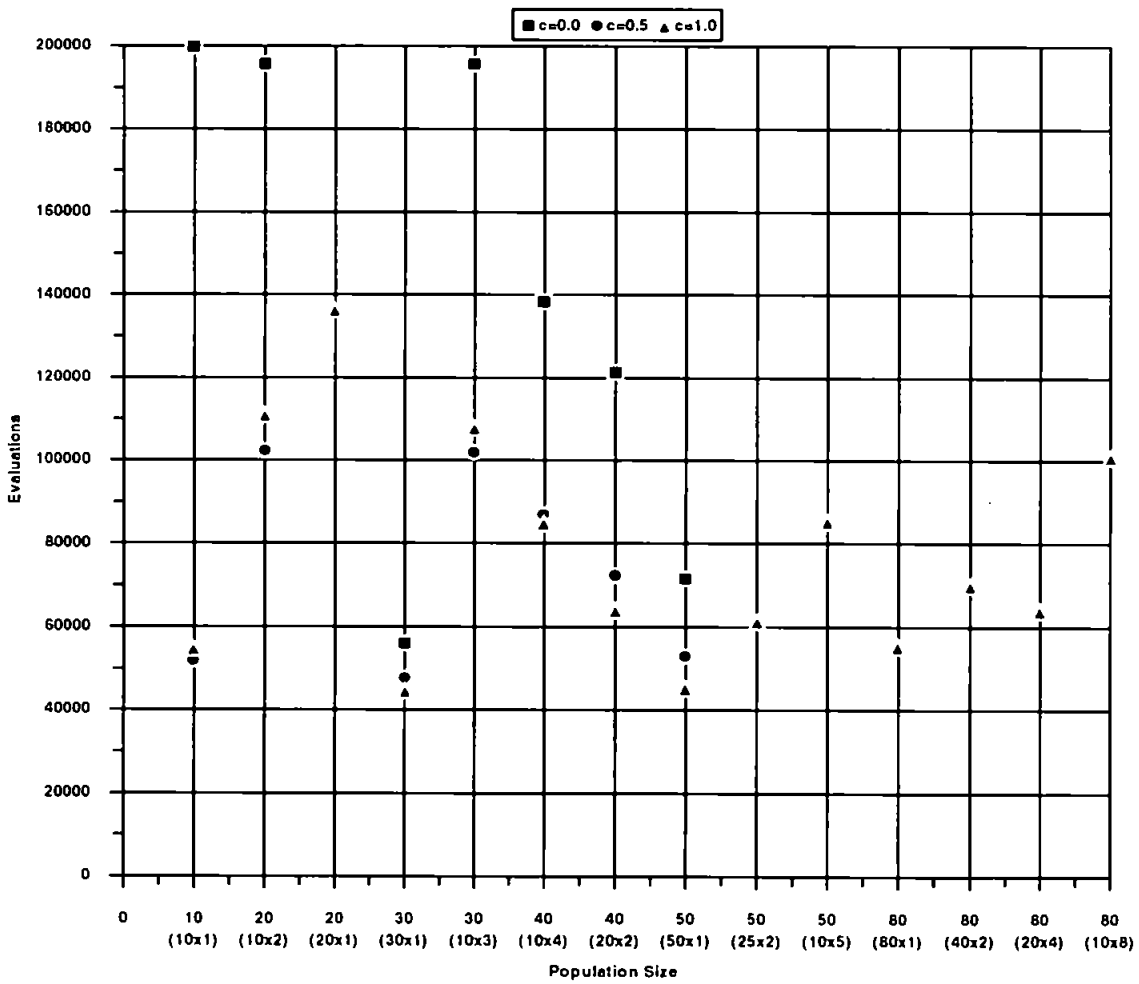


Figure 5.12- Two-Box Results

Figure 5.12 indicates that the best results are achieved when the crossover mutation rate is set to 1.0. The minimum amount of computational effort required is produced with a population size of 50x1 with 45,185 individuals needing to be processed. This compares with 3320 iterations using NN's and 1,176,000 evaluations using standard GP with a population size of 4000.

5.7.2 Complex Multiplication

The previous example of symbolic regression involved one or more independent variables, but only one dependent variable. The problem described here is that of multiple regression. This problem attempts to find the unknown relationships between two independent variables, y_1 and y_2, and four dependent variables, x_1 ,x_2 ,x_3 , and x_4 given 50 six-tuples of data. Where the target function is vector multiplication, i.e.:-

$$y_1 = x_1 x_3 - x_2 x_4 \quad \text{and} \quad y_2 = x_2 x_3 - x_1 x_4.$$

The DRAM-GP parameters for this problem are shown in table 5.48.

Functional set	F = { +, -, *, %, LIST2 }
Arguments	F_A={ 2 , 2, 2, 2 }
NC Functionals	N_F={1.1, 1.1, 1.2, 1.2 }
Terminal set	T = { x1 , x2 , x3 , x4 }
NC Terminals	N_T={ 1.0, 1.0, 1.0, 1.0}
Test points (TP)	50
Fitness	MSE
Max. NC	50.0
Min. NC	0.0
Elite	5
Crossover mutation rate	1.0
CCC	± 2.0 of NC value
Max. Chromosome length	100
Max. generations	200

Table 5.48 - Run Parameters For Complex-Multiplication Problem

The functional LIST2 performs no function other than linking two sub-trees. It is used only once at the top of each tree and allows the whole tree to evolve the two independent variables. The results of various runs are presented in table 5.49.

	Pop. size M (pop.x.species)	Effort E
Koza, 1992 (STD)	500	609,500
DRAM-GP	10 (10x1)	729,810
DRAM-GP	20 (10x2)	229,500
DRAM-GP	30 (30x1)	90,720
DRAM-GP	50 (50x1)	97,000
DRAM-GP	100 (20x5)	100,100
DRAM-GP	150 (30x5)	155,700
DRAM-GP	200 (40x5)	142,800
DRAM-GP	250 (50x5)	155,000

Table 5.49 - Complex-Multiplication Results

5.7.3 Simple Symbolic Regression Problem

Suppose a sampling of the numerical values from a target curve over 20 points in some domain is given, such as the real interval [-1.0,+1.0]. That is, a sample of data in the form of 20 pairs (x_i , y_i) is given, where x_i is the value of the independent variable in the interval [-1.0,+1.0] and y_i is the associated value of the dependent variable. The 20 values of x_i are chosen at random in the interval [-1.0,+1.0]. The target function here is the polynomial expression:

$$y = x^4+x^3+x^2+x \qquad (5.14)$$

The goal is to find a function, in symbolic form, that is a good or a perfect fit to the 20 pairs of numerical data points.

The terminal set is

$$T=\{x\}. \qquad (5.15)$$

The next step is to identify the set of functions that are used to generate the mathematical expressions that attempt to fit the given finite sample of data. If knowledge that the answer is $x^4+x^3+x^2+x$ is used, a function set consisting of only addition and multiplication operations would be sufficient for this problem. A more general choice might be the functional set consisting of the four ordinary arithmetic operators of addition, subtraction,

multiplication, and the protected division function %. If a wider variety of problems are to be solved, the sine function SIN, the cosine function COS, the exponential function EXP, and the protected logarithm function RLOG are included. The functional set for this problem is thus:

$$F=\{ +, -, *, \%, SIN, COS, EXP, RLOG\} \quad (5.16)$$

Taking two,two,two,two,one,one,one,one arguments respectively.

The raw fitness for this problem is the sum, taken over the 20 fitness cases, of the absolute value of the difference (error) between the value in the real-valued range space produced by the expression for a given value of the independent variable x_i and the correct y_i in the range space. The closer this sum is to zero, the better the computer program. Error-based fitness is the most common measure of fitness used.

The hits measure for this problem counts the number of fitness cases for which the numerical value returned by the expression comes within a small tolerance called the hits criterion of the correct value. For this example, the hits criterion is 0.01 (Koza, 1992).

Published computational effort result (Koza, 1992) using a population size M of 500 and a maximum number of generations of 50, is 162,500 evaluations. The results are based on 113 runs. The parameters for the DRAM-GP run are shown in table 5.50.

Functional set	F = { +, -, *, %, SIN, COS, EXP, RLOG }
Arguments	F_A={ 2, 2, 2, 2, 2, 2, 2, 2 }
NC Functionals	N_F={1.1, 1.1, 1.2, 1.2,1.3,1.3,1.4,1.4 }
Terminal set	T = { x }
NC Terminals	N_T={ 1.0 }
Test points (TP)	20
Fitness	MSE
Max. NC	50.0
Min. NC	0.0
Elite	5
CCC	± 2.0 of NC value
Chromosome length	100
Max. generations	200

Table 5.50 - DRAM-GP Parameters For The Simple Symbolic Regression Problem

The results for the simple symbolic regression problem are shown in tables 5.51 and 5.52. The computational effort is based on 100 runs for each table.

C=1.0 I=100 P=20x4	
% runs	Effort E
90%	6,080

Table 5.51 - Simple Symbolic Regression Problem With A Population Size Of 80

C=1.0 I=100 P=50x10	
% runs	Effort E
90%	15,000

Table 5.52 - Simple Symbolic Regression Problem With A Population Size Of 200

The results show that with a population size of 80 individuals only 6,080 evaluations are required to produce a correct solution, this compares with 162,500 evaluations using standard GP and a population size of 500 (Koza, 1992).

5.8 Continuous Symbolic Regression Problems

The previous example of symbolic regression contained no numerical constants within the target curves or within the terminal set, nor was there any explicit facility for creating them. The search space of possible solutions was discrete, although large, but with the inclusion of real numbers the search space is infinite.

5.8.1 Symbolic Regression Using Real Numbers

Real numbers are now included in the continuous equation. The test function here is:-

$$y = 0.5\ x^2 \tag{5.17}$$

The functional set is F = { +, -, *, % }with arguments { 2, 2, 2, 2 } respectivly. The terminal set has to include x and the set of real numbers $\Re$. When a program tree is being created and a terminal is selected there is a 50% chance that a real number will be chosen. A range is set for these real numbers which for this problem is between -5.0 and 5.0 and the number used is randomly chosen between these limits. If a real number is chosen for mutation it is replaced by another randomly selected number. The parameters for the DRAM-GP run are shown in table 5.53.

Functional set	F = { +, -, *, % }
Arguments	F_A={ 2, 2, 2, 2}
NC Functionals	N_F={1.1, 1.1, 1.2, 1.2 }
Terminal set	T = { x , $\Re$ }
NC Terminals	N_T={ 1.0 }
Test points (TP)	20
Fitness	MSE
Max. NC	20.0
Min. NC	0.0
Elite	5
CCC	± 2.0 of NC value
Max. Chromosome length	100
Max. generations	200

Table 5.53 - DRAM-GP Parameters For The Symbolic Regression Problem $0.5x^2$

Tables 5.54 and 5.55 show the results of 100 runs for each parameter set.

C=1.0 I=100 P=20x4	
% runs	Effort E
71	19,410

Table 5.54 - $0.5x^2$ Symbolic Regression Problem With A Population Size Of 80

C=1.0 I=100 P=50x4	
% runs	Effort E
100	13,000

Table 5.55 - $0.5x^2$ Symbolic Regression Problem With A Population Size Of 200

The results of Koza (Koza, 1992) state that using a population size of 200, 19,800 evaluations are required to solve the problem with a probability of success of 99%, based on 190 runs.

One evolved solution to this problem included no real numbers, but did include the sub tree x/(x+x) which reduces to the real number 0.5. This shows that even if real numbers are not explicitly included within the GP process, real numbers can, and are produced. Due to the nature of the test function the generation of the required real number is relatively easy to produce, so, to further examine if the revised GP method can be used with real numbers a more complex symbolic regression problem is used.

5.8.2 Increased Complexity Symbolic Regression

The test problem is $y = 2.718\ x^2 + 3.1416\ x$ and 20 equally spaced test points are produced between the range of $-1.0 < x < 1.0$. It would be much more difficult for the numbers Pi and e to be generated without the use of real numbers. The range of any created (or mutated) real numbers is again between -5.0 and 5.0. The fitness is calculated using 'hit points' (Koza, 1992) a hit being scored if the y value of the evolved solution is within 0.01 of the test value.The parameters for the DRAM-GP run are shown in table 5.56.

Functional set	F = { +, -, *, % }
Arguments	F_A={ 2 , 2, 2, 2}
NC Functionals	N_F={1.1, 1.1, 1.2, 1.2 }
Terminal set	T = { x , ℜ }
NC Terminals	N_T={ 1.0 }
Test points (TP)	20
Max. NC	40.0
Min. NC	0.0
Elite	5
CCC	± 2.0 of NC value
Max. Chromosome length	705
Max. generations	50

Table 5.56 - DRAM-GP Parameters For The Complex Symbolic Regression Problem

Tables 5.57 to 5.60 show the results of 10 runs for each parameter set.

C=1.0 I=160 P=20x8	
% runs	Effort E
10	225,280

Table 5.57 - Complex Symbolic Regression Problem With A Population Size Of 160

C=1.0 I=250 P=50x5	
% runs	Effort E
30	85,800

Table 5.58 - Complex Symbolic Regression Problem With A Population Size Of 250

C=1.0 I=300 P=50x6	
% runs	Effort E
80	29,700

Table 5.59 - Complex Symbolic Regression Problem With A Population Size Of 300

C=1.0 I=500 P=50x10	
% runs	Effort E
80	38,400

Table 5.60 - Complex Symbolic Regression Problem With A Population Size Of 500

The results compare with 305,500 evaluations using a population size of 500, based on 100 runs (Koza, 1992).

5.9 Summary

The results show that performance improvements similar to that obtained using ADF's (Koza, 1992,1994) is possible using DRAM-GP, but the population size and thus the memory required to initiate a run is greatly reduced. ADF's look for sub-trees which can be repeated within the tree structure and so will perform well on problems which have solutions which can be constructed from these fit sub-trees. This is shown in the 5 parity problem where ADF's outperformed DRAM-GP. Future research could explore the use of DRAM-GP with ADF's for the Boolean induction class of problems.

Although all performance calculations assume that the computational effort required to evaluate a single generation is the same for conventional GP, GP using ADF's, and RAM-GP, it is evident that this is not the case with DRAM-GP. The crossover operator requires extra administration to find sub-trees with NC values within the required range. However this extra administration cost is considered small when compared with the fitness function computational effort and therefore of little consequence.

The distribution of NC values throughout the individuals will be biased towards lower values (i.e. lower NC values will have a higher frequency than higher ones), and so because the sub-tree selection is random for CCC, there is a higher probability that the sub-tree to be replaced will have a low NC value. The smaller sub-trees will also have less chance of being disrupted by the mutation operator within the CCC operator. Individuals are created and are then *slowly* reduced or expanded in size through the action of the CCC operator. Once one species finds a solution which is fitter than any other individual within the total population, it very quickly propagates this information to other species, accelerating the evolution towards fitter solutions.

If a run is continued after an exact solution is found, the evolutionary process will continue to evolve solutions that have lower NC values and thus less complex solutions.

The injection mutation, IM, is important for the DRAM-GP paradigm in that it naturally disrupts the fitness of the individuals within each species (apart from the elite 5) and then evolves solutions using the elite and disrupted individuals. This prevents the population from prematurely converging and can be considered as keeping the population in a state of entropy, from which new fitter individuals are created. When the injection mutation is

turned off the species quickly converge, and the performance of the adaptive program is greatly reduced.

CHAPTER 6

APPLICATIONS TO PRELIMINARY DESIGN SOFTWARE

Chapter five has shown that DRAM-GP outperforms conventional GP in terms of both performance and population size. For problems involving discrete search spaces, GP is the most effective search method for systems identification of all the evolutionary computation techniques. However the GP paradigm has no efficient method for searching continuous search space as are encountered when real numbers are included within the problems under investigation. GP produces real numbers by initially randomly creating them within the initial population and then through the action of crossover the real numbers are manipulated to produce more real numbers, a process that is random and also one which adds a computational overhead to the problem solving process.

The new GP technique presented is thus extended to incorporate *both* continuous and discrete search spaces as is found within the domain of preliminary design. The revised technique is called **HDRAM-GP** (Watson & Parmee, 1997(b)) i.e. **H**ybrid **D**istributed, **R**apid, **A**ttenuated **M**emory, **G**enetic **P**rogramming. **HDRAM-GP** incorporates a real numbered genetic algorithm to aid search in the continuous space. Its application is demonstrated on engineering fluid dynamics systems which were initially investigated in section 4.5.1.

6.1 A Hybrid Extension To DRAM-GP

For mixed discrete and continuous search spaces, an algorithm which will efficiently explore through both spaces is required. In order to achieve this **HDRAM-GP** uses two alternating crossover operators. The GP operator searches the discrete functional structure whilst the

GA searches the continuous coefficient space. The GP crossover operator can select parents from any species and is thus called an inter-species crossover operator, while the GA crossover operator is limited to selecting parents from the same species and is thus intra-species. This crossover regime is possible due to the adopted computer representation of the individuals i.e. an array of characters for the functional description and an array of floating point numbers for the real number coefficients. GP crossover then operates within the character array to evolve the functional structures of the system, whilst GA crossover concurrently operates only within the real number array to evolve the coefficients of the structures.

The GA crossover operator is restricted to individuals within the same species and *only* manipulates the real numbers stored in the floating point array within the individual structures. Two parents, P1 and P2 are randomly selected from the same species and a single crossover point, CP1, is selected. AS in standard GP crossover this defines a sub-tree which is to be crossed. Only the real numbers of P1 and P2 are swapped with all functionals remaining unchanged. The resulting child individual is then evaluated.

The GA crossover thus produces only one new individual and so is performed twice for every GP crossover, thus ensuring both operators produce the same amount of children. The HDRAM-GP algorithm performs one standard GP crossover, directly followed by two GA style crossovers, this cycle is repeated until the correct number of crossover operations has taken place.

Every equivalent generation mutation occurs and changes only one allele within each individual with a probability of mutation of 0.5 The top 5 individuals are elite and are never mutated, but are allowed to participate in crossover.

6.2 Explicit Formula For Friction Factor In Turbulent Pipe Flow

This problem was first investigated in section 4.5.1 and a brief summary of the results produced is shown. The initial functional in every individual was set to $\log_{10}$ this reduced the problem to finding the sub-function y in the following equation:-

$$f^{-0.5} = a.\log_{10} y \qquad (6.1)$$

where a=constant and $y = f(RE, K/D)$.

Using the following functional and terminal sets.

$$F = \{ +, -, *, \% \} \qquad (6.2)$$

and $$T = \{ Real, Re, K/D \} \qquad (6.3)$$

The best evolved formula using standard GP with a local hill climber,presented in section 4.5.1,

is: -

$$\frac{1}{\sqrt{f}} = -3.8364 \log_{10}\left\{\frac{0.2097K}{D} + \frac{11.1001}{\mathrm{Re}}\right\} \qquad (6.4)$$

The average error being within 0.27% of Colebrook and White's formula. This required a total of 1,000,000 evaluations to solve the problem. The inclusion of the hill climber increases computational expense, requiring a total of 3 runs, seeding successive runs with a simplified equation of the best result from the previous run.

Using HDRAM, with a population size of 40 consisting of 4 species of 10 individuals, and 1000 generations, the best evolved equation from 5 independent runs is:-

$$\frac{1}{\sqrt{f}} = 3.7922 \log_{10}\left\{-15.0169 - \left[\frac{-5.4353}{\left(57.3353/\left(717.8244 + \mathrm{Re}\right)\right) + K/D}\right]\right\} \qquad (6.5)$$

with an average error of 0.79%. Although the result is less accurate than the standard GP/hill climber technique, HDRAM requires only one run to provide a satisfactory solution and computational expense is reduced due to the much smaller population size and the exclusion of the local hill climber.

A second series of runs assumed a functional form for the resulting equation of :-

$$f^{-0.5} = y \tag{6.6}$$

where $y = f(Re\ ,\ K/D\ )$

This increases the complexity of the problem by including the log_{10} operator in the functional set. Using the standard GP/hill climber technique with a population size of 1000, and 1000 generations, performance is poor with runs producing average errors in the range of 20% to 30%.

The best result of 5 independent runs using HDRAM-GP with a population size of 10 individuals in 4 species, is:-

$$\frac{1}{\sqrt{f}} = 3.50391 log_{10}\left\{\frac{3.2917Re}{18.7046 + K/D\,(0.41644 - Re)}\right\} \tag{6.7}$$

The average error being within 1.82% of Colebrook and White's formula. The reduced accuracy of the result is due to the increased complexity of the problem domain but HDRAM-GP shows a significant improvement over standard GP/hill climber techniques for this increased complexity problem.

Run parameters are shown in table 6.1.

Number of species	4
Species population size	50
Maximum NC value	40
Minimum NC value	0
Crossover mutation rate	0.5
Maximum chromosome length	750
Real number minimum value	-5.0
Real number maximum value	5.0
Maximum generations	1000
Elite	5

Table 6.1 - Run Parameters For Friction Factor Problem

The results of 5 independent runs are shown in table 6.2.

Run no.	Worst error (%)
1	5.73
2	6.81
3	2.12
4	2.94
5	2.52

Table 6.2 - Results For Friction Factor Problem

It can be seen that HDRAM-GP not only produces acceptable results using fewer function evaluations, but can also derive equations without a prespecified functional form.

6.3 Laminar Two-Dimensional Sudden Expansion Flow Problem

Previous work on sudden expansion flow in section 4.5.2 showed that although the standard GP paradigm can model this system to some extent, it requires multiple runs to achieve a satisfactory solution. The results were obtained by reducing the dimensions of the problem for an initial run i.e. only using the data at Re=1000, and seeding a second run with the best result of the first run, but using the whole range of test data for Re= 100,200,...,1000.

Using **HDRAM-GP**, equations are evolved which describe the X and Y velocities using only one run of the algorithm and all the data i.e Re=100,200,...,1000. The resulting equation can produce velocity vectors for Re in the range 0→1000. Figures 6.1 and 6.2 show the X and Y velocity components at Re=1000.

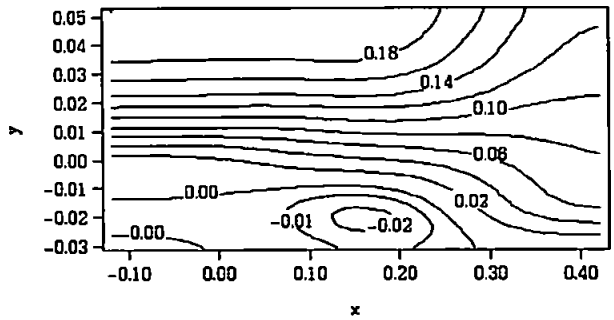


Figure 6.1 - X-Velocity Test Surface

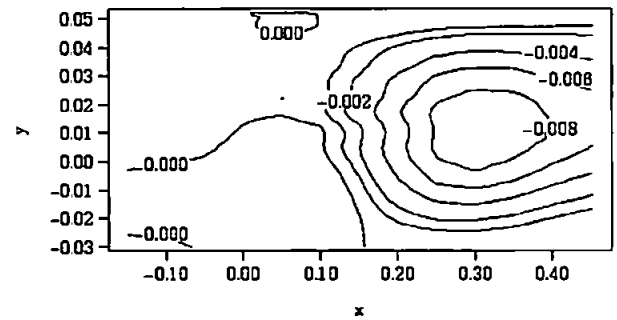


Figure 6.2 - Y-Velocity Test Surface

HDRAM is used with a population size of 400, consisting of 10 species of 40 individuals within each species, and 1000 generations. Equations are evolved which describe the X and Y velocities using only one run of the algorithm and all of the test data i.e. Re=100,200,...,1000. This again shows that HDRAM-GP can solve more complex problems than standard GP.

Figures 6.3 and 6.4 show the evolved X and Y velocity at Re=1000 of the best of 5 independent runs. The average error is 15.5% and again the improved technique has a reduced computational expense, and produces acceptable solutions using only one run of the algorithm.

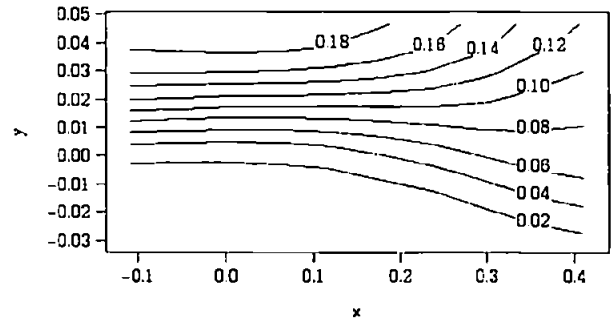


Figure 6.3 - Evolved X-Velocity Surface

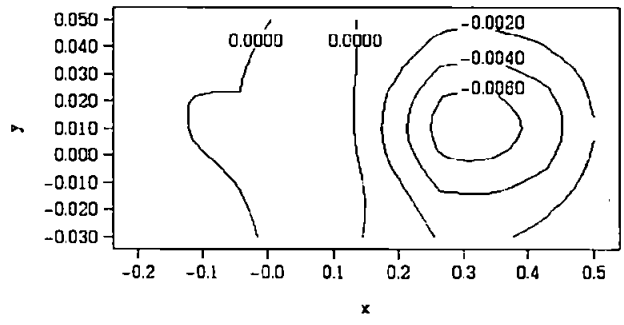


Figure 6.4 - Evolved Y-Velocity Surface

HDRAM-GP outperforms DRAM-GP in several ways. The population size required to solve the problem is reduced from 10,000 individuals to 400 using HDRAM-GP. HDRAM-GP had to initially solve the problem at a fixed Reynolds number (a reduced dimension) and then required further runs using the results from the initial run as a population seed to produce equations which represented the flow over the range of Reynolds numbers given in the test data.

6.4 Thermal Paint Jet Turbine Blade Data

The final problem under investigation is the modelling of the surface temperature of a turbine blade under set operating conditions, first investigated in section 4.5.3. Figure 6.5 shows a typical surface of a turbine blade used for this test.

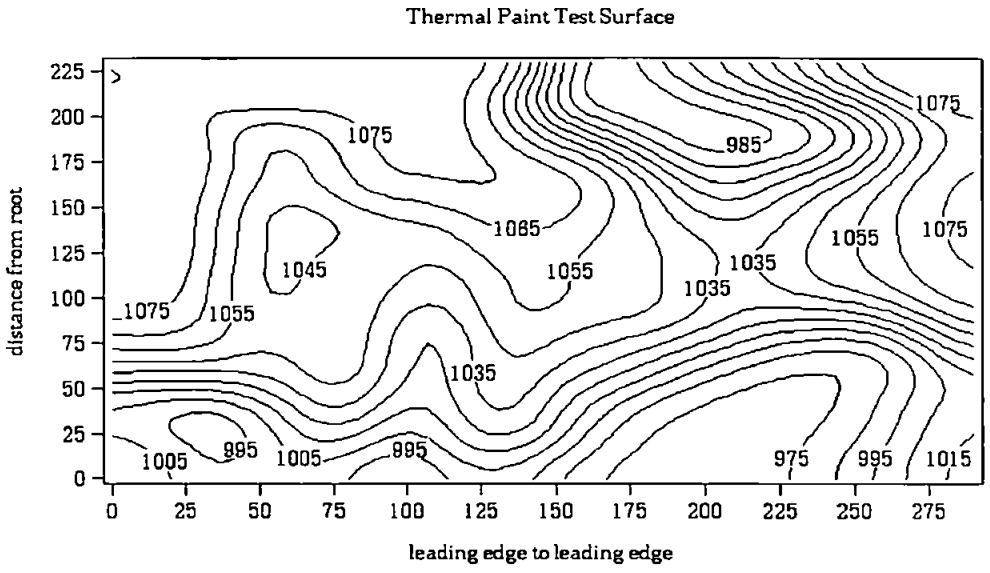


Figure 6.5 - Thermal Paint Test Surface

This problem was investigated using standard GP in section 4.5.3 and acceptable results have been produced for one-dimensional curves. The best result for the modelling of the whole surface (i.e. surface fitting) using the standard GP/hill climber approach has an average error of 13.4% using a population size of 1000 and a total of 10,000 generations.

The functional set and terminal set are

Figure 6.6 shows the best evolved surface from 5 independent runs using HDRAM using a population size of 400 individuals consisting of 40 species of 10 individuals and 1000 generations.

The evolved surface has an average error of 7.5% over the 290 test points used. This again shows that the technique outperforms the standard GP/hill climber method with a greatly reduced population size and smaller number of generations and thus fewer fitness evaluations.

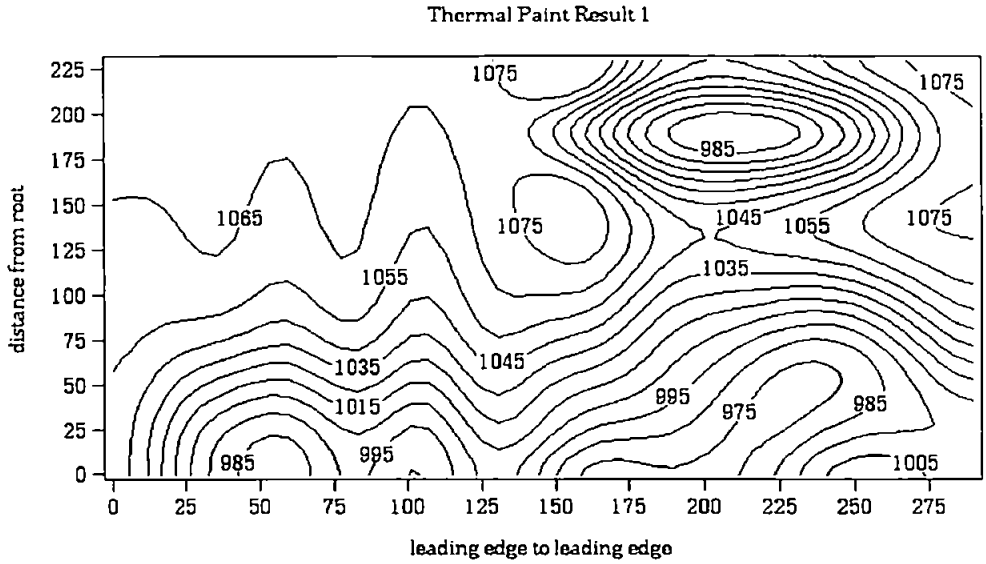


Figure 6.6 - Evolved Thermal Paint Surface

6.5 Summary

The results presented here show that HDRAM-GP can be applied to model approximate equations to engineering systems with better or at least comparable accuracy to that of earlier standard GP/hill climber methods. Advantages of this technique compared with a standard GP/hill climber approach are:-

- A reduction of overall population size and required generations.
- Reduced CPU running time.
- A reduction in computer memory required to run the evolutionary program.
- The ability to search discrete structures and continuous coefficients concurrently.
- A control mechanism for the lengths of individual tree structures.
- Ability to search higher dimensional problems.
- An efficient method for searching for numerical values.

The GP crossover mechanism is responsible for the transfer of information from species to species, and fit solutions are rapidly transferred to other species, including those of a lower NC grouping species, thus producing less complex solutions. The node complexity measure, which controls the GP crossover, minimises disruption by ensuring some similarity between crossed sub-trees and also controls tree length growth. The injection mutation disrupts the fitness of the individuals within each species (apart from an elite 5 individuals within each species) and then evolves solutions using the elite and disrupted individuals. The method can also model systems which are too complex for conventional genetic programming and the hybrid GP/hill climber technique.

CHAPTER 7

DISCUSSION

At the beginning of this thesis the objectives of the research were stated as follows:-

1) To identify the utility of evolutionary computation and in particular genetic programming for systems identification.
2) To develop appropriate evolutionary strategies for systems identification.
3) The integration of complementary adaptive search and traditional optimisation techniques for systems identification..
4) The improvement of areas of simple engineering software using the developed strategies.

The first objective was dealt with in chapters 2 and 3 with a comparison of various techniques presented in chapter 4. From the investigation into the utility of different methods of evolutionary computation it has been shown that genetic programming provides the best method in terms of representation. The inputs to GP are usually presented directly in terms of the observed variables of the problem domain. Therefore, the representation used by genetic programming is the natural representation of the problem domain. The lack of pre-processing is a major distinction relative to conventional genetic algorithms operating on strings, neural networks, and other machine learning algorithms. Neural networks provide a mathematically proved method for solving *any* problem, the major drawback being that the results of the network are virtually impossible to view and represent as a mathematical function. Genetic programming can provide interpretable equations and does not require any prior knowledge of the system as in the case of a GA.

The second objective was achieved only after conventional GP was used on various problems. The results of these initial runs showed that GP produces solutions that are very long and the lack of any mechanism for the evolution of numerical values required the inclusion of a local hill-climber to search for the correct numerical values. These drawbacks to conventional GP were then addressed by the use of a new GP algorithm called DRAM-GP, and the success of this technique has been clearly demonstrated. The problem of numerical constant manipulation still existed with DRAM-GP and so to solve problems in both the discrete and continuous search spaces a hybrid technique has been produced called HDRAM-GP thus achieving objective three. This technique has been shown, through experimentation, to be better then conventional GP for solving systems identification problems. Finally chapter six used HDRAM-GP to evolve models for use within the engineering design domain.

For HDRAM-GP the crossover mechanism is responsible for the transfer of information from species to species, and fit solutions are rapidly transferred to other species, including those of a lower NC grouping species, thus producing less complex solutions. The node complexity measure, which controls the GP crossover, minimises disruption by ensuring some similarity between crossed sub-trees and also controls tree length growth. The injection mutation disrupts the fitness of the individuals within each species preventing the population from prematurely converging. When the injection mutation is turned off the species quickly converge, and the effectiveness of the technique is greatly reduced.

The concurrent utilisation of the GA reduces the amount of processing required to obtain acceptable results by further reducing semantic disruption. The method can also model systems which are to complex for conventional GP and the hybrid GP/hill climber technique.

7.1 Conclusions

The results presented here show that HDRAM-GP can be applied to model approximate equations to engineering systems with better accuracy to that of earlier conventional GP and GP/hill climber methods. Advantages are:-

- A reduction of overall population size and required generations.
- Reduced CPU running time.
- A reduction in computer memory required to run the evolutionary program.
- The ability to search discrete structures and continuous coefficients concurrently.
- A control mechanism for the lengths of individual tree structures.

The research has shown that HDRAM offers a better potential for the modelling of selected engineering systems producing improved calibrations when compared to standard GP/hill climber methods.

The initial testing of conventional GP on engineering systems also showed that GP does not scale up to higher dimension problems very well. This was shown with the turbine blade data in which only 2-dimensional curves could be fitted with any degree of accuracy, and also with the pipe friction factor problem where a functional form had to be initially set to enable the GP algorithm to solve the problem, thus reducing the search space to a level that could be readily be solved by the application of conventional GP. With HDRAM-GP these problems were overcome but although the new method can solve problems up to 4 dimensions, it is the author's view that GP does not scale up past 4 dimensions.

There is also a continuing debate about the action of the crossover operator within GP. Some researchers believe that crossover is a macro-mutation operator (Angeline, 1997), and it is the author's view that this is true, but only for certain problem domains, systems identification being one of these. If crossover is responsible for the transmission of 'good' genetic material then the use of the crossover mutation rate within HDRAM-GP should be set to zero. This was the case with Boolean induction problems, but when problems which contain continuous search spaces are encountered the best reported results occur when the crossover mutation rate is set to 0.5 or 1.0. This suggests that the crossover operator used on continuous problems is indeed actually a mutation operator.

7.2 Future Research Directions

The research reported in this thesis has shown the ability of GP to solve simple systems identification problems, but the research also opens other areas relating to GP and systems identification, possible future research is outlined here.

One area that requires further work is that of addressing the problem of scalability of GP. It has been shown that by modifying the conventional genetic programming algorithm, higher dimensional problems can be solved, but the new technique extends the problem solving by only a few dimensions. There seems to be no method for solving systems identification problems which include real numbers with more than 5 dimensions. Further research could explore the utility of HDRAM-GP together with a local hill-climber and also the inclusion of ADF's. It has been shown that the hill-climber can improve the fitness of individuals and also ensure that fit functional structures are not lost due to poor terminal selection. The inclusion of ADF's could increase the performance of HDRAM-GP but the choice of problem would have to accommodate repeated sub-trees.

Other theoretical work on schemata theory for GP has been attempted but the work makes assumptions about the crossover operator and mutation method, a more general GP schemata theory would help researchers understand the mechanisms that make GP work effectively.

REFERENCES

Andre, A. & Teller A.; (1996) A Study In Program Response And The Negative Effects Of Introns In Genetic Programming. 1st Conference on Genetic Programming. Morgan Kaufmann.

Angeline, P.J.; (1997) Subtree Crossover: Building Block Engine Or Macromutation. 2nd International Conference on Genetic Programming. Morgan Kaufmann.

Asimow, M. (1962). Introduction to Design. Prentice-Hall, Inc., Englewood Cliffs, N.J.

Badekas, D. & Knight D.D. (1992). Eddy Correlations For Laminar Axisymmetric Sudden Expansion Flows. Journal of Fluids Engineering, Vol. 114.

Bagley, J.D. (1967). The behavior of adaptive systems which employ genetic and correlation algorithms. Ph.D. Dissertation, University of Michigan, Ann Arbor, MI.

Banzhaf, W.; Nordin, P.; & Francone, F. D. (1997) Why Introns In Genetic Programming Grow Exponentially. Workshop on Exploring Non-coding Segments and Genetics-Based Encodings at ICGA 1997. Michigan State University, MI.

Bickel, A.S. & Bickel, R.W. (1987). Tree Structured Rules in Genetic Algorithms. Proc. Of the 2nd International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates. P77-81.

Borowski, E.J. and Borwein, J.M. (1989). Collins dictionary of Mathematics. Harper Collins.

Chellapilla, K. (1997). Evolutionary Programming With Tree Mutations: Evolving Computer Programs Without Crossover. 2nd International Conference on Genetic Programming. Morgan Kaufmann.

Cox, M. G. (1990). Algorithms For Spline Curves And Surfaces. NPL Report DITC 166/90.

Cramer, N.L. (1985). A Representation For The Adaptive Generation Of Simple Sequential Programs. Proceedings of the 1st International Conference on Genetic Algorithms. Erlbaum.

Dannon, Y. (1993). WinNN (version 9.0). Public Domain Neural Network Shareware. Troy, NY 12180. Email:danony@rpi.edu

Darwin, C. (1859). On the Origin of Species by Means of Natural Selection. John Murray.

Dasgupta. D. (1991). A Structured Genetic Algorithm. Research Report IKBS-2-91, University of Strathclyde, UK.

Dasgupta. D. (1992). Nonstationary Function Optimisation Using Structured Genetic Algorithms. Proceedings of Parallel Problem Solving From Nature (PPSN 2). Springer-Verlag.

Davis, L., Ed. (1991). Handbook of Genetic Algorithms. Van Nostrand Reinhold, New York, NY.

De Jong, K.A. (1975). An analysis of the behavior of a class of genetic adaptive systems. Ph.D. Dissertation, University of Michigan.

D'haeseleer, P. (1994). Context Preserving Crossover In Genetic Programming. IEEE.

Douglas, J.F.; Gasiorek, J.M. & Swaffield, J.A. (1985) Fluid Mechanics (2nd Edition). Longman Scientific & Technical, John Wiley & Sons, New York.

Eberhart, R. Simpson, P. & Dobbins, R. (1996). Computational Intelligence PC Tools. Academic Press Inc.

Fogel, L.J. (1994). Evolutionary programming in perspective: the top-down view. Computational Intelligence: Imitating Life, J.M. Zurada, R.J. Marks II and C.J. Robinson, Eds., IEEE Press, Piscataway, NJ.

Fraser, A. S. (1957). Simulation of genetic systems by automatic digital computers. Australian Journal of Biological Science, 10:484-499.

Fraser, A. S. (1960). Simulation of genetic systems by automatic digital computers: 5-linkage, dominance and epistasis. Boimetrical Genetics, O.Kempthorne, Ed., Macmillan, New York, NY, 70-83.

Fraser, A. S. (1962). Simulation of genetic systems. Journal Of Theoretical Biology, 2:329-346.

Friedberg, R.M. (1958). Alearning machine: Part I. IBM journal of Research and Development, 2:2-13.

Friedberg, R.M., B. Dunham, and J.H. North (1959). A learning machine: Part II. IBM journal of Research and Development, 3:282-287.

Fujiki, C. (1986). An Evaluation of Holland's Genetic Algorithm Applied to a Program Generator. M.S. thesis, Department of Computer Science, University of Idaho.

Fujiki, C. & Dickinson, J. (1987). Using the genetic algorithm to generate LISP source Code To Solve The Prisoner's Dilemma. Proceedings of the 2nd International Conference on Genetic Algorithms. Erlbaum.

Goldberg, D.E. (1983). Computer-aided gas pipeline operation using genetic algorithms and rule learning. Ph.D. dissertation, University of Michigan.

Goldberg, D.E. & Richardson, J. (1987).Genetic Algorithms With Sharing For Multimodal Function Optimization. Proceedings of 2nd International Conf. On Genetic Algorithms.

Goldberg, D.E. (1989). Genetic Algorithms in Search, Optimisation, and Machine Learning. Addison-Wesley, Reading, MA.

Goldberg, D. E. (1991(a)) Real-coded Genetic Algorithms, Virtual Alphabets, and Blocking. Complex Systems 5. 1991.

Goldberg, D.E., Deb, K. & Korb, B. (1991(b))Don't Worry, Be Messy. Proceedings of 4th International Conf. On Genetic Algorithms.

Goldberg, D.E. & Rudnick M. (1991(c)). Genetic Algorithms and the Variance of Fitness. Complex Systems 5: p265-278.

Goldberg, D. E.; Deb, K. & Clark, J.H. (1992). Genetic Algorithms, Noise, and the Sizing of Populations. Complex Systems 6:p333-362.

Goldberg, D.E., Kalyanmoy, D., Kargupta, H. & Harik, G. (1993). Rapid, Accurate Optimization Of Difficult Problems Using Fast Messy Genetic Algorithms. IlliGAL Report No. 93004.

Haaland, S.E. (1983). Simple and explicit formulas for the friction factor in turbulent pipe flow. Journal of Fluids Engineering 105: p89-90.

Haynes, T. Phenotypical Building Blocks for Genetic Programming. 2nd Annual Conference Genetic Programming (GP'97) MIT Press, 1997.

Hicklin, J. (1986). Application of the Genetic Algorithm to Automatic Program Generation. M.S. thesis, Department of Computer Science. Moscow, ID: University of Idaho.

Holland, J.H. (1962). Outline for a logical theory of adaptive systems. Journal of the Association for Computing Machinery, 3:297-314.

Holland, J.H. (1975). Adaptation in natural and artificial systems. Ann Arbour. The University of Michigan Press.

Iba, H., DeGaris, H. & Sato, T. A. (1993). Systems identification using structured genetic algorithms. Proceedings of the 5th international conference on genetic algorithms: p279-286. San Mateo, CA: Morgan Kaufmann. 1993.

Iba, H., DeGaris, H. & Sato, T. A. (1996(a)). A Numerical Approach to Genetic Programming for System Identification. Evolutionary Computation 3(4): p417-452.

Iba, H. (1996(b)). Random Tree Generation For Genetic Programming. Proceedings of the 1st Annual Conference on Genetic Programming. Stanford, CA. Stanford University.

Jiang, M. (1992). A Hierarchical Genetic System For Symbolic Function Identification. Masters Thesis. University of Montana.

Jiang, M. Wright, A.H. A Hierarchical Genetic System For Symbolic Function Identification. Proceedings of the 24th Symposium on the Interface: Computing Science and Statistics, College Station, Texas.

Jiang, M. (1993). An Adaptive Function Identification System. Proceedings of the IEEE/ACM Conference on Developing and Managing Intelligent Systems Projects, Vienna, Virginia.

Johnson, T. & Husbands, P. (1991). Systems Identification Using Genetic Algorithms. Parallel Problem Solving From Nature: p85-89. Springer-Verlag.

Jones,T. (1995). Crossover, Macromutation, and Population Based Search. Proceedings of the 6th international conference on genetic algorithms: p73-80. Cambridge, MA: Morgan Kaufmann. 1993.

Kargupta, H. & Smith, R. E. (1992). Systems Identification with Evolving Polynomial Networks. Proceedings 4th international conference on genetic algorithms: p370-376.

Kinnear Jr. K. E. (1993). Generality and difficulty in Genetic Programming: Evolving a Sort. Proc. of 5th International Joint Conference on Genetic Algorithms.

Koza, J.R. (1992). Genetic Programming: On the Programming of Computers by means of Natural Selection. MIT Press, Cambridge, MA.

Koza, J., Keane, M.A. & Rice, J.P. (1993). Performance Improvements Of Machine Learning Via Automatic Discovery Of Facilitating Functions As Applied To A Problem Of Symbolic Systems Identification. IEEE International Conference On Neural Networks.

Koza, J.R. (1994). Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, MA.

Lancaster, P.; Salkauskas, K. (1986). Curve and surface fitting. Academic Press.

Langdon, W.; (1997). Fitness causes Bloat. 2nd Annual Conference Genetic Programming (GP'97) MIT Press, 1997.

Messa, K. (1992). Fitting Multivariate Functions To Data Using Genetic Algorithms. Proceedings of Society Of Photo-Optical Instrumentation Engineers.

Ninomiya, H. & Onishi, K. (1991). Flow analysis using the PC, CRC Press, Inc.

O'Reilly, U-M. & Oppacher, F. (1995). The Troubling Aspects Of A Building Block Hypothesis For Genetic Programming. 2nd Conference on Genetic Programming. Morgan Kaufmann. 1997.

Pandya, A. S.; Macy, R. B. (1995). Pattern Recognition with Neural Networks in C++. CRC Press/IEEE Press.

Parmee, I.C. (1996). The Development Of A Dual-Agent Strategy For Efficient Search Across Whole System Engineering Design Hierarchies. Proceedings Parallel Problem Solving from Nature. (PPSN IV), Lecture notes in Computer Science No. 1141 : p523-532. Springer-Verlag, Berlin.

Pahl G., Beitz W. (1988) Engineering Design - A Systematic Approach. English Edition, The Design Council.

Poli, R. & Langdon, W.B. (1997). A New Schema Theory For Genetic Programming With One-Point Crossover And Point Mutation. 2nd Conference on Genetic Programming. Morgan Kaufmann.

Rechenberg, I. (1965). Cybernetic solution path of an experimental problem. Royal Aircraft Establishment, library translation 1122, Farnborough, Hants, U.K.

Rechenberg, I. (1973). Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Frommann-Holzboog Verlag, Stuttgart, Germany.

Schaffer, J.D.; Caruana, R.A.; Eshelman, L.J. & Das, R. (1989). A study of control parameters affecting online performance of genetic algorithms for function optimization. Proccedings of the 3rd International Conference on Genetic Algorithms.

Schaffer, J. D. & Eshelman, L. J. (1991). On Crossover as an Evolutionarily Viable Strategy. Proceedings of 4th International Conference On Genetic Algorithms.

Schwefel, H.P. (1965). Kybernetische Evolution als Strategie der experimentellen Forschung in der Stromungstechnik. Diploma thesis, Technical University of Berlin, Germany.

Soule T., Foster,J.A., Dickinson, J.; (1996) Code Growth In Genetic Programming . 1st Conference on Genetic Programming. Morgan Kaufmann.

Spears, W.M. & Anand, V. (1991).Study Of Crossover Operators In Genetic Programming. Methodologies For Intelligent Systems. 6th International Symposium, ISMIS Proceedings.

Spect, D.F. 1990. Probabilistic Neural Networks. Neural Networks, 3(1): p108-118.

Syswerda, G. (1989). Uniform Crossover In Genetic Algorithms. Proceedings of the 3rd International Conference on Genetic Algorithms. San Mateo, California. Morgan Kaufmann

Tenorio,M.F. & Lee, W. (1990). Self-Organising Network for Optimum Supervised Learning. IEEE Transactions on Neural Networks. Vol 1. No.1. p100-110.

Watson, A. H. & Parmee, I. C. (1996), Identification Of Fluid Systems Using Genetic Programming. Proceedings EUFIT '96. Vol I, p395-399. ELITE Foundation.

Watson, A.H. & Parmee, I.C. (1997). Steady State Genetic Programming with Constrained Complexity Crossover Using Species Sub-Populations. Procs. 7th International Conference on Genetic Algorithms. (ICGA 97). Morgan Kaufmann.

Welstead, S.T. (1994). Neural Network and Fuzzy Logic Applications in C/C++. Wiley.