Cranfield UNIVERSITY

An Investigation Of Genetic Algorithms And Genetic Programming

Ibrahim KOLCU

School of Industrial and Manufacturing Science

MPhil Thesis

1996

ProQuest Number: 10820910

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10820910

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by Cranfield University.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Cranfield UNIVERSITY

School of Industrial and Manufacturing Science

MPhil Thesis

1994-1996

Ibrahim KOLCU

An Investigation Of Genetic Algorithms and Genetic Programming

Supervisor: Dr. Alfred VELLA

This dissertation is submitted in partial fulfilment of the requirements for the degree of *Master of Philosophy in Computer Science*

An Investigation Of Genetic Algorithms And Genetic Programming

Ibrahim KOLCU

June, 1996

Abstract

There are many regression techniques that try and fit known models to sets of data. For them we assume the functional form of the model and use analytic or statistical techniques to find the values of any unknown model parameters.

If the target model is thought to be of sufficiently complex a form, the above techniques (i.e. analytic and statistical techniques) may fail to provide the desired results and alternative methods have to be used. This is even more important if the underlying model is itself unknown.

Genetic algorithms and genetic programming are two techniques that may help in the search for suitable models. Unfortunately, however, both of these techniques have themselves parameters that need to be specified and there are no clear guidelines to aid such choice. A number of other implementation issues are also open questions and in this thesis we look at a number of ways of implementing genetic algorithms and genetic programs to evaluate alternatives.

Simple target models are used throughout most of this work so that the effects of changes to the method's parameters can be monitored.

We look at how population size, crossover probability and mutation rate affect the speed of convergence of the genetic algorithm to an acceptable model.

One of the most difficult aspects of genetic programming is the issue of the meaning of the offspring produced by crossover or mutation. Some systems arrange that any offspring that do not have meaning are removed from the

population. Others ensure that no such offspring can arise. In this work we look at what might happen if we always impose a meaning on all possible offspring.

In the genetic programming part of this work we look at two representations of our models. In the first we used a fixed length representation, whilst in the second we used a tree to represent each member of the population.

We also look at a number of fitness functions. The commonest such functions are based upon errors between the model and the data. For our fitness functions we also use their correlation coefficient. We found that a strategy that starts by using correlation coefficient and then a fitness that combines both correlation coefficient and error worked better.

Acknowledgements

First and foremost, I thank The God, Who gave me the strength and opportunity to complete this study successfully.

A special acknowledgement goes to my supervisor, Dr. Alfred Vella. He gave me the necessary inspiration, and his help made a great contribution in preparation of the thesis.

My deep gratitude also goes to the reviewers for their critical and constructive suggestions. They are Carol Vella, Nick Benwell and Carole Peachy-May.

I am in debt to Mustafa Karakoy and Hilal Al-Ismaily for their encouragement and support whenever needed during the time I spent at Cranfield.

Contents

Abstract i
Acknowledgements iii

1. Introduction 1
1.1. Research objectives 2
1.2. Symbolic regression 4

2. Genetic algorithms & genetic programming 5
2.1. Introduction 6
2.2. Genetic algorithms 6
2.2.1. Background 6
2.2.2. Representation 8
2.2.3. Creation of the initial population 9
2.2.4. Fitness 9
2.2.6. Reproduction 11
2.2.7. Crossover 12
2.2.8. Mutation 13
2.2.9. An example of a simple genetic algorithm approach 14
2.3. Genetic programming 19
2.3.1. Background 19
2.3.2. Representation 22
2.3.3. Creation of the initial population 23
2.3.4. Fitness 24
2.3.6. Reproduction 25
2.3.7. Crossover 25
2.3.8. Mutation 27

3.Computer implementation of GAs&GP 29
3.1. Introduction 30
3.2. Data structures for GAs 30
3.3. Reproduction, crossover, and mutation for GAs 33
3.4. Calculation of fitness 36
3.5. Output of the GAs 37
3.6. Data structures for genetic programming 38

4. Modelling & experimental design 41
4.1. Introduction 42
4.2. Modelling the systems 42
4.2.1. The first model for genetic algorithms (GAs-1) 42
4.2.2. The second model for genetic algorithms (GAs-2) 44

4.2.3. The first model for genetic programming (GP-1) 45
4.2.4. The second model for genetic programming (GP-2) 49
4.3. Experimental design 50
4.3.1. The design for experiments using GAs-1 & GAs-2 50
4.3.2. The design for experiments using GP-1 55
4.3.3. The design for experiments using GP-2 57

5. Discussion on experimental results 59
5.1. Introduction 60
5.2. The data sets used in the experiments 60
5.2. Solution space 63
5.3. The importance of the fitness function 64
5.4. The effect of selecting a population size 67
5.5. Crossover probability 69
5.6. Mutation probability 70
5.7. Testing the model GAs-1 72
5.8. Testing the model GAs-2 82
5.9. Testing the model GP-1 94
5.10. Testing the model GP-2 118

6. Symbolic regression for strain on a helicopter's main rotor 132
6.1. Introduction 133
6.2. Problem definition 133
6.3. Evolutionary search algorithms and helicopter health and usage monitoring 136
6.4. Flight parameter data 136
6.5. Experimental design 139
6.6. Results 141

7. Conclusions 143
7.1. Introduction 144
7.2. Discussion on the models 144
7.3. Discussion on the helicopter experiment 147
7.4. Further work 148
7.5. Conclusion 149

References 150

CHAPTER 1

INTRODUCTION

1.1. RESEARCH OBJECTIVES

Discovering relationships between sets of data is a task which has exercised researchers' mind since the dawn of modern science. It can be laborious and error prone, and it is highly subject to researcher's bias. Because many of today's research problems are more complex than those of the past, it is increasingly important that robust data analysis methods are available to researchers. In order for a data analysis method to be most useful, it must meet at least three criteria: good predictive ability, insight into the inner workings of the system being analysed, and unbiased results.

Historically, researchers deduced relationships solely by examining the data - a difficult task if the relationship is complex, if many variables are involved, or if the data are noisy (as often occurs in real-world problems). Moreover, the examination is easily influenced by the researcher's desires and expectations (Siegel&Morgan 1996).

Statistical methods were among the first tools developed to help a researcher find the relationships of observed facts. Statistical methods are often based on such assumptions as: (1) the data are normally distributed, (2) the equation relating the data is of a specific form (for example, linear, quadratic, or polynomial), and (3) the variables are independent (Hays 1988). If the problem meets these assumptions, statistics can be a valuable tool for finding the relationships. But real-world problems seldom meet these criteria fully.

Neural networks, an artificial intelligence technique, are not limited by these assumptions. They serve as strong predictive models that can uncover complex relationships, but they give little insight into the underlying mechanisms that describe a relationship. However, two other AI techniques, Genetic

Algorithms (GAs) and Genetic Programming (GP), are more robust methods for exploring complex solution spaces. Independently, they have had some success at revealing the mechanisms relating data items

Recently, genetic algorithms, which use the principles of evolution through natural selection to solve problems, have established themselves as a powerful search and optimisation technique. Basic GAs proceed as follows:

(1) Create a population of random individuals, in which each individual represents a possible solution to the problem at hand.
(2) Evaluate each individual's fitness - its ability to solve the specified problem.
(3) Select individual population members to be parents.
(4) Produce children by recombining parent material via crossover and mutation, and add them to the population.
(5) Evaluate the offspring's fitness.
(6) Repeat steps 3-5 until a solution with the desired fitness goal is obtained.

Researchers have successfully applied GAs to many problems (Goldberg 1989, 1994). Genetic algorithms and their use are thoroughly discussed by Goldberg as a problem-solving and function optimisation technique (Goldberg 1989). Although linear GAs are adept at developing rule-based systems, they cannot develop equations. A recent addition to the evolutionary domain is genetic programming (Koza 1992), which uses an evolutionary approach to generate symbolic expressions and perform symbolic regression.

In this research, genetic algorithms and genetic programming concepts were investigated in relation to symbolic regression. In an attempt to assess GAs' and GP's performance, they were used to find relationships between

components' loads within helicopters and the corresponding flight data. This approach requires both the load components and corresponding flight data (e.g. positions of controls, aircraft accelerations, etc.). Once a function which represents the relationship is found, it is possible to use the function to predict load components without using strain gauges and/or slip ring load measuring equipment.

1.2. SYMBOLIC REGRESSION

Symbolic regression (i.e. 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 independent variables. That is, symbolic regression involves finding a model that fits a given sample of data (Koza 1992).

When the variables are real-valued, symbolic regression involves finding both the functional form and the constants for the model. Symbolic regression differs from conventional linear, quadratic, or polynomial regression, which merely involve finding the constants for a function whose form has been prespecified.

In any case, Genetic Algorithms (GAs) and Genetic Programming (GP) can be used to perform symbolic regression. GP can find the form of the function and the associates (Koza 1992). In order to optimise the coefficients of a function, GAs can be used (Goldberg 1989).

CHAPTER 2

GENETIC ALGORITHMS & GENETIC PROGRAMMING

2.1. INTRODUCTION

In this chapter, the literature review which has been done on Genetic Algorithms and Genetic Programming is discussed.

2.2. GENETIC ALGORITHMS

2.2.1. BACKGROUND

A genetic algorithm (GA) is a directed random search technique, first developed by John Holland at the University of Michigan. His research outlined the process of natural selection and genetic evolution as an analogy for search and optimization. During the process, biological adaptive systems strive to optimize single individuals or entire species to increase survivability within dramatically changing environments. GAs are simulations of the methods used when biological systems adapt to their environment supplied in computer software models to solve optimization and machine learning problems.

The basic frameworks of genetic algorithms inspired from natural adaptation are:

1. A single individual changes within its environment to promote survival. The way we learn - the neural changes that accompany learning - is an example of how an individual adapts within its environment.
2. During this adaptation, the fitter members of a population have more chance to reproduce.

(Bayer & Wang 1991)

Search techniques used to solve problems in engineering, science, economics, finance, etc. can be classified as calculus-based techniques or direct-search techniques. If problems are well defined, either numerically or analytically, or

the solution space is unimodal and small, calculus-based techniques perform well. However, calculus-based techniques perform poorly on the broad spectrum of optimisation functions that arise in engineering optimisation. Many direct-search techniques, which in general require only function information (i.e. experimental results), have been suggested to overcome the shortcomings of calculus-based techniques. The main problem of direct-search methods is that they require a trade-off between available computer time and exactness of the optimal solution (Goldberg 1989).

GAs are efficient search procedures that can identify optimal or near-optimal solutions from solution populations. GAs iteratively apply genetically-inspired operators to solution populations, generating new, fitter populations whilst searching for optimal or near-optimal solutions to the problem in hand. They are very effective where other search techniques perform poorly. GAs are well suited to problems with a large search space.

GAs have gained popularity among engineers from varying fields in recent years as a robust optimisation tool for variety of problems. They were first used by Holland and his associates in developing machine learning systems, and then later analysed and extended by De Jong, Goldberg, and others (Goldberg 1989, KrishnaKumar 1992). For example, Goldberg describes an application of GAs to develop a system for controlling a gas pipeline that consists of large numbers of branches and valves to regulate the flow of gas. He also highlights their use in areas such as machine learning, pattern recognition, image-processing and operational research (Goldberg 1989, Stockton 1993).

GAs are different from normal search procedures in four ways (Goldberg 1989):

1. GAs work with a coding of the parameter set, not the parameters themselves (i.e. an individual in a population represents a set of parameters).
2. GAs search from a population of points, not a single point.
3. GAs use objective function information, not derivatives or other auxiliary knowledge.
4. GAs use probabilistic transition rules, not deterministic rules.

Genetic algorithms use probabilistic-rule set to manipulate the parameters of the objective function which are coded as a finite-length string. GAs are also blind, that is, they know nothing about the problem being solved except the fitness function. Genetic algorithms perform iteration by iteration, generating and testing a population of individual strings. Population is a key word here. They try to increase the fitness of the members of the population.

A simple genetic algorithm that has obtained good results is made up of three operators:

1. Reproduction.
2. Crossover.
3. Mutation.

These three operators are explained in the following text.

2.2.2. REPRESENTATION.

Before applying GAs to a task, a representation of possible solutions must be found. The most common representation uses fixed length bitstrings (Janson&Frenzel 1993). In the classic genetic algorithm, each individual in the population is represented by a fixed-length binary string of an individual that

encodes the problem parameter set. This encoded string can be decoded to give for example decimal values for these parameters.

In Figure 2.1, an individual with fixed length representation is shown.

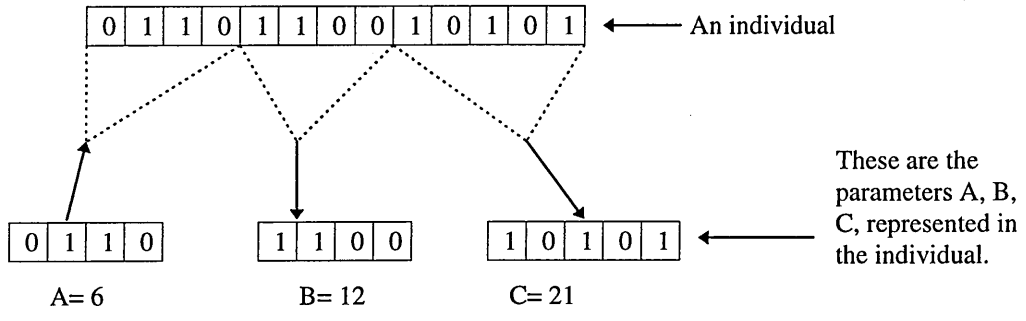


Figure 2.1. An example 13 bit length representation.

2.2.3. CREATION of the INITIAL POPULATION

The initial population consists of a set of potential solutions generated randomly or heuristically (Bayer & Wang 1991). The first method, creating the initial population randomly, is preferred for problems where no knowledge exists about the solutions. The second method involves employing a *priori* knowledge about the given problem. This knowledge is used to form an initial population, so that GAs start from these approximately known solutions and thus may help to reach an optimal solution in less time than the previous method (Pham & Liu 1995).

2.2.4. FITNESS

Once the initial population has been created, the evaluation phase begins. Genetic algorithms require that members of the population can be

measure of how well a particular member of the population solves the given problem. It is measured by decoding the problem parameters and using them as input to a fitness function. The members that are fitter are given a higher probability of being selected in the reproduction, crossover, and mutation phase.

For some problems, the fitness of an individual can be measured using the 'error' between the result produced from the individual and the desired result. The closer this error is to zero, the better the individual. This error is usually measured as a sum (or average) over a number of representative combinations of the inputs to the problem (i.e. values of an independent variable) (Figure 2.2). The correlation coefficient between the desired answer and the output produced from the individual may be used in calculating the fitness (Koza 1994).

$$F = \frac{1}{1 + \underbrace{\sum_{i=1}^{N} \| R_{pi} - R_{di} \|}_{E}}$$

F : Fitness value of an individual.
R_{pi} : Result produced from an individual.
R_{di} : Desired data.
N : Number of data in a set.
E : Sum of the absolute errors between R_{pi} and R_{di}.

Figure 2.2. An example of a fitness function. Here the fitness value is always between 0 and 1.

2.2.6. REPRODUCTION

The reproduction operation involves selecting an individual from the current population of individuals on the basis of its fitness and allowing it to survive by copying it into the new population (Goldberg 1989). Selection models nature's survival-of-the-fittest mechanism (Srinivas & Patnaik 1994).

In the reproduction process, individuals are copied according to their objective function values (i.e. fitness function). The fitness function can be thought of as some measure of profit or goodness that needs to be increased as much as possible. Every individual in the population space generates a number of copies decided on the basis of its fitness, with the best individuals producing many copies, and the worst not producing any. This is the process that allows GAs to take advantage of a survival-of-the-fittest strategy.

There are several ways of calculating the number of offspring generated by GAs. To avoid premature convergence, a proportional selection method may be used (Pham&Liu 1995). One simple method of implementing reproduction is "roulette wheel selection" (Figure 2.6). Fitness values of individuals represent the widths of slots in the wheel. After a random spinning of the wheel to select an individual for the next generation, slots with large widths representing individuals with high fitness values will have a higher chance of being selected. This method of selecting gives a chance for individuals with a lower fitness to be selected (Goldberg 1989). In this method, the probability of an individual being able to reproduce is given by Figure 2.3.

$$P_{iselect} = \frac{F_i}{\sum_{i=1}^{N} F_i}$$

(F_i : the fitness value for the i_{th} individual)

(N: number of individuals)

Figure 2.3. The formula used for the calculation of the probability of individual to reproduce.

2.2.7. CROSSOVER

A crossover operation is used to create new individuals (offspring) from two parental individuals selected on the basis of their fitness. Crossover is the process by which any set of randomly paired strings exchange their substring information. Two individual parents are mated randomly and they exchange substring information at a random position in the string, between 1 and the string length minus one. A decision is made to perform crossover depending on the selected value of the probability of crossover, P_c, usually between 0.5 and 1.0 (Goldberg 1989). This process of creating new strings, i.e. solutions, is a powerful mechanism for improving the fitness values of individual solutions.

If two individuals are somewhat effective in solving a given problem, then some of their parts probably have some merit. Crossing over these substrings may well yield a new individual that is more effective at solving the problem (Koza 94). An example crossover operation in binary string representation is shown in Figure 2.4.

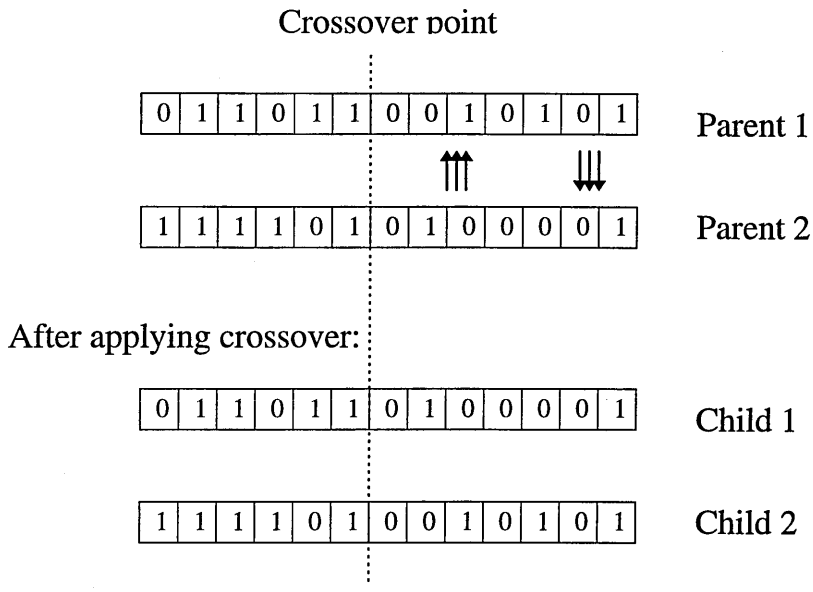


Figure 2.4. An example crossover operation in the binary string representation.

2.2.8. MUTATION

Mutation is an operation involving the random alteration of a bit in a string and is a secondary mechanism of GAs. It is used to keep the population from prematurely converging on one, sub-optimal solution, and to make new genetic material. In a binary code, this involves changing a 1 to a 0 and vice versa. The value of the probability of mutation P_m is usually kept very low say in the range 0.001 to 0.01 (Deshpande at al 1992).

Figure 2.5 shows a mutation operation in bitstring representation.

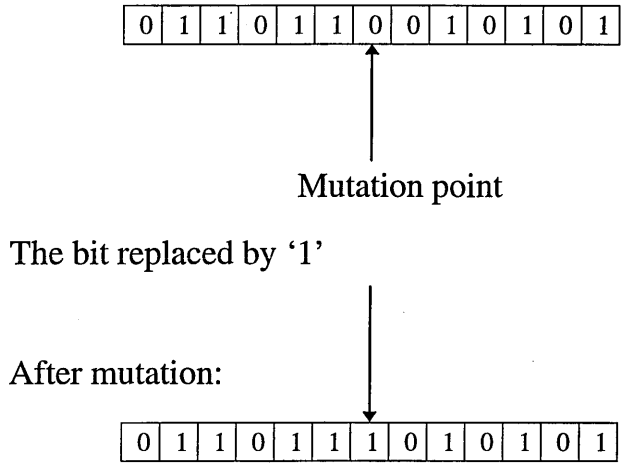


Figure 2.5. An example of mutation operation in binary string representation

2.2.9. AN EXAMPLE of A SIMPLE GENETIC ALGORITHM APPROACH

A simple genetic algorithm is applied to the problem of maximising the function $f(x)= x^2$, where x is an unsigned integer value between 0 and 31. The function $F(x)= x^2$ is the fitness function.

In order to code the decision variable x as a finite-length string a binary unsigned integer of length 5 is required.

0 (0 0 0 0 0) 31 (1 1 1 1 1)

Firstly an initial population is needed and it can be generated randomly or heuristically. Four individuals are selected (i.e. N=4) by tossing a coin 20 times.

The fitness values are calculated from the string representation by decoding an individual string as an unsigned binary integer (i.e. x).

string $\rightarrow$ 0 1 0 0 0 decoding $\rightarrow$ x= $2^3 = 8$ $f(x)= x^2 = 8^2 = 64$

A generation of the GAs begins with reproduction. The expected number of copies are calculated by 'N*$P_i$select' as shown in Table 2.1. Thus, the best get more copies, the average stay even and the worst die off.

During the crossover process, strings are mated randomly via the roulette wheel. A random point along the string is selected as the crossover point and then, mated string couples cross over.

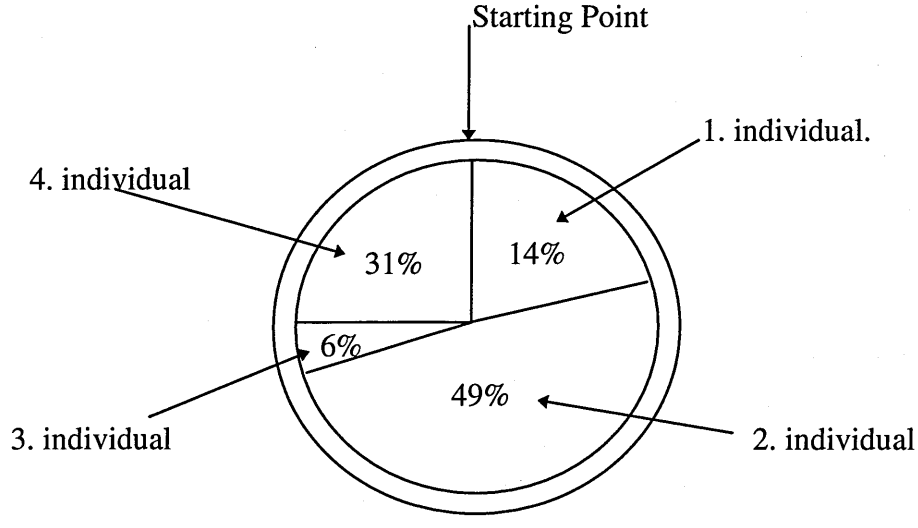


Figure 2.6. The roulette wheel for the example.

P_iSelect (fitness) (Table 2.1) values of the individuals represent the widths of slots on the roulette wheel. In the selection, a real number between 0 and 1 is produced randomly. For example if it is 59%, the second individual will be selected because 59% is greater than 14% and less than 63% (=14%+49%). (Figure 2.6).

Here, the crossover probability is assumed to be P_c =1.0.

The results are shown in Table 2.2.

The last operator, mutation, is performed on a bit-by-bit basis. In this example, the probability of mutation is $P_m = 0.001$.

The resulting population is in Table 2.2 as a new population and the average fitness can be seen to have increased from 293 to 439 in one generation.

A brief flow chart of the genetic algorithm can be seen in Figure 2.7.

String No	Initial Population	x Value	F(x) x2	PiSelect	N x Piselect	copies
1	0 1 1 0 1	13	169	0.14	0.56	1
2	1 1 0 0 0	24	576	0.49	1.96	2
3	0 1 0 0 0	8	64	0.06	0.24	0
4	1 0 0 1 1	19	361	0.31	1.24	1
Sum			1170	1.00	4.00	4.0
Average			293	0.25	1.00	1.0
Max			576	0.49	1.96	2.0

Table 2.1. Reproduction operator.

Mating Pool after Reproduction (Cross Site Shown)	Mate (Randomly Selected)	Crossover Site (Randomly Selected)	New Population	x	F(x) x2
0 1 1 0 | 1	2	4	0 1 1 0 0	12	144
1 1 0 0 | 0	1	4	1 1 0 0 1	25	625
1 1 | 0 0 0	4	2	1 1 0 1 1	27	729
1 0 | 0 1 1	3	2	1 0 0 0 0	16	256
Sum					1754
Average					439
Max					729

Table 2.2. Crossover operator.

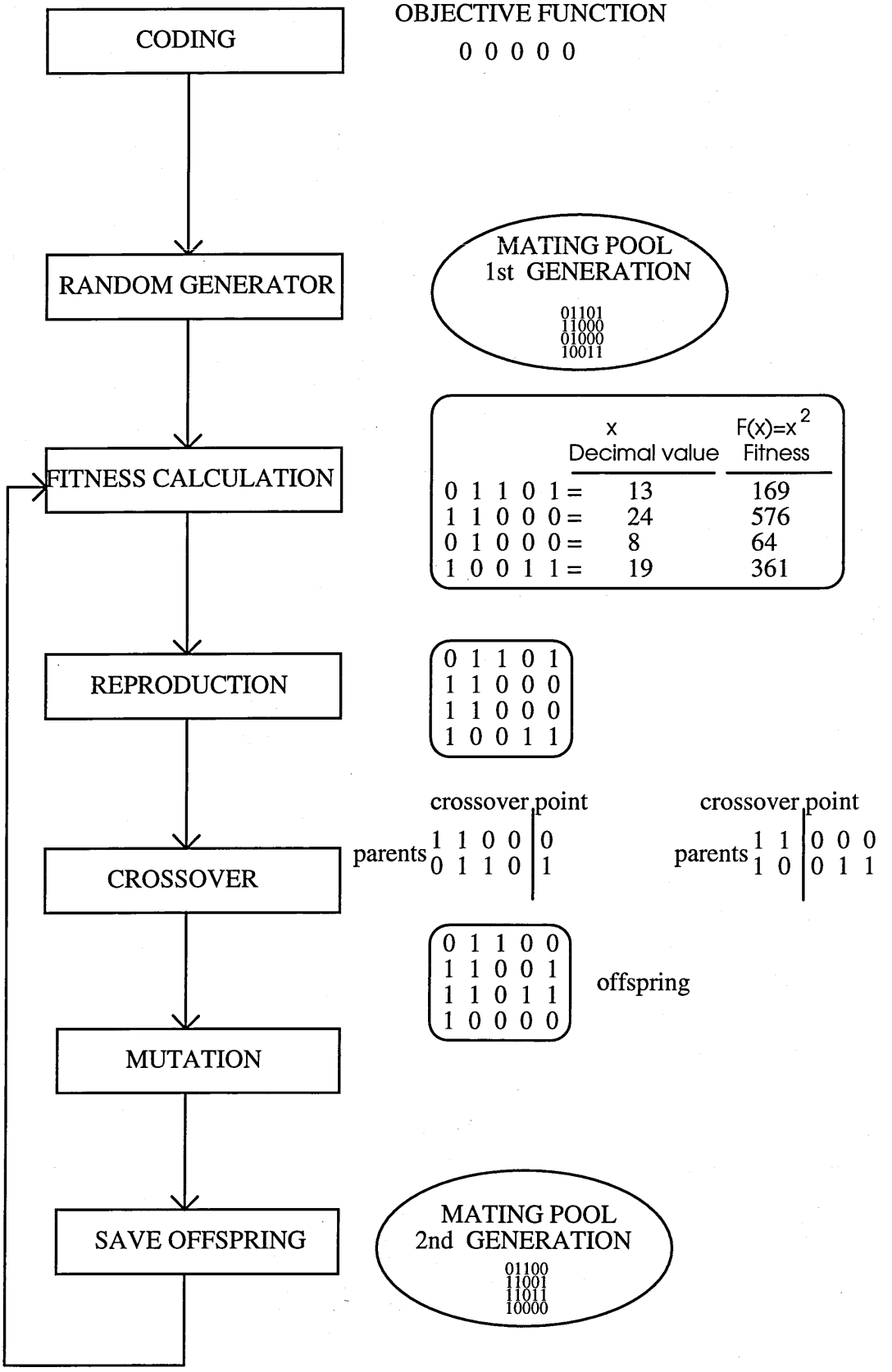


Figure 2.7. An example Genetic Algorithm.

2.3. GENETIC PROGRAMMING

2.3.1. BACKGROUND

Genetic programming is a technique using the mechanism of the genetic algorithms in which the structures undergoing adaptation are hierarchical computer programs, and an attempt to get computers to learn to solve problems without being explicitly programmed (Koza 1992).

According to Kenneth in 1993, the technique was first proposed by John Koza in 1989 and is described and illustrated in Koza's book (Koza 1992). The application of GP to a particular data set has the advantage over neural networks in that it appears to be able to discover the underlying relationship between data sets, and to specify this in a functional form, albeit with a certain amount of redundancy (Smith 1994). The technique uses the mechanisms of genetic algorithms, but applies them to populations of LISP programs. Although Koza used LISP, computer languages such as C, Pascal can be used with an appropriate data structure for GP. The LISP programs and the trees which represent them are constructed from a function set (e.g. sin, tan, +, -, *, /) and a terminal set (i.e. parameters for the functions or constants) chosen by the user. A function takes one or more arguments, which may be terminals or may themselves be functions. Functions form the root and the internal nodes of the tree. Terminals form the leaves and may represent input variables or constants. Fitness-proportionate selection is carried out in the same manner as that of conventional GAs. In the GP crossover operation, randomly selected subtrees of each chosen parent pair are swapped to form a pair of syntactically correct offspring. Mutation consists of randomly replacing subtrees in existing members of the population with new randomly generated subtrees (Koza 1994).

Individuals which are composed of functions and terminals appropriate to the problem domain are called computer programs in genetic programming. An example computer program is shown in Figure 2.6.

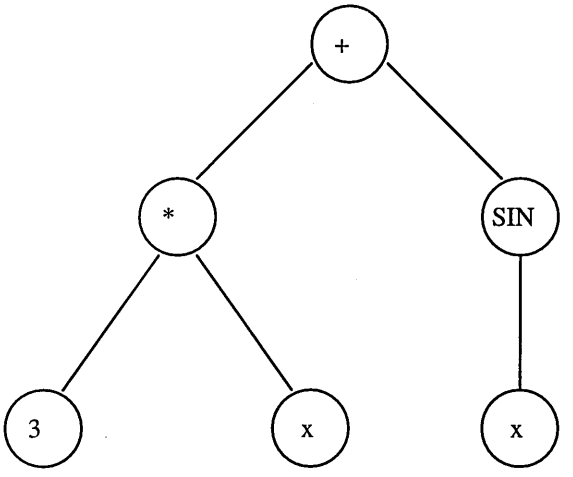


Figure 2.6. A computer program.
$(3*x)+SIN(x)$

According to Koza, there are five preliminary steps to solving a problem using genetic programming (Koza 1994).

These are choosing

(1) the terminals,

(2) the functions,

(3) the fitness function,

(4) the control parameters,

(5) and the termination criterion.

The first step in applying genetic programming is to identify the terminal set for the problem. The terminals correspond to the inputs of the computer program.

The second step is to identify the function set. The functions are standard arithmetic operations, standard mathematical functions, or domain-specific functions.

In Koza's terminology, the terminals and the functions are the elements of the programs. In Figure 2.6 functions '+', '*', and '*SIN*' are used to form the junctions in the tree. The terminals, which form the end leaves, may be either variables or constants. In Figure 2.6 '*x*', '*3*' are the terminals of the computer program. The connections between the terminals and functions indicate the order in which operations are to be performed.

The fitness function defines the fitness of an individual computer program as a solution for the required problem. In most cases the goal is to find an individual with the maximum fitness. The choice of the terminals, the functions and the fitness function largely determine the space which genetic programming searches.

The control parameters include issues such as the size of population, the crossover probability, the mutation probability and the maximum size of the programs.

The termination criterion is simply a rule for stopping. Typically the rule is to stop either when a program which solves the problem is found or after a given number of generations, e.g. 50.

2.3.2. REPRESENTATION

In genetic programming, the GAs' fixed length representation of individuals is replaced by different sized programs. Usually a tree representation of the program is used and individuals in the population are of different sizes.

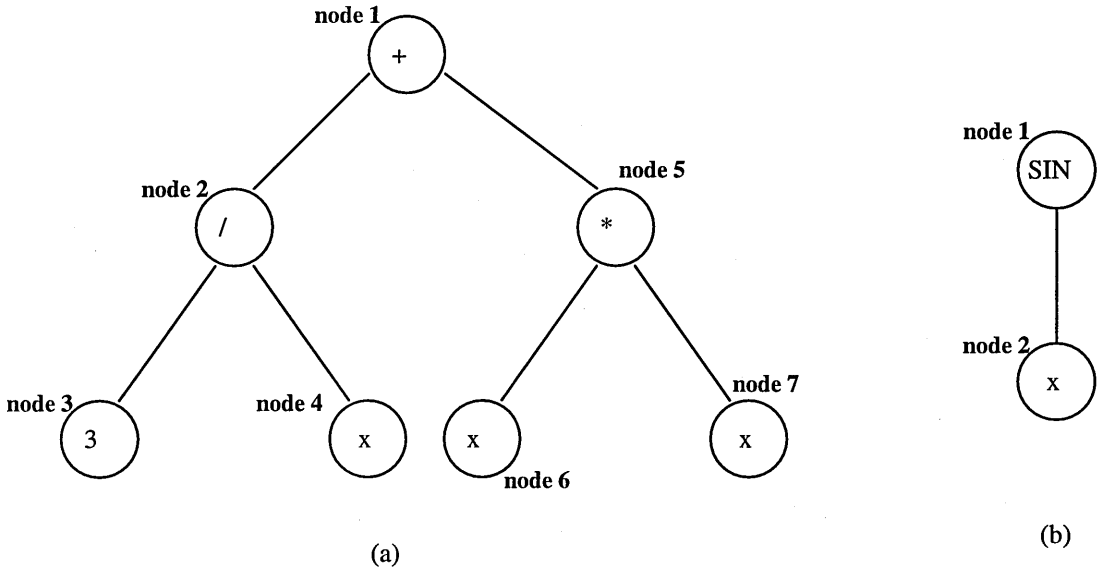


Figure 2.7. Two different individuals of different sizes in a population..

a. $(3/x)+(x*x)$

b. $SIN(x)$

In this tree representation, every node represents either a terminal or a function. The nodes at the end of connections are called terminals, as they terminate the connections of the tree. A terminal may be a constant or one of the inputs (variables) of the problem which GP will try to solve. The other nodes are called functions, as they are functions that accept parameters and pass a value up to the next level. The parameters accepted by the functions will

either come directly from the terminals, or from other functions lower down the tree. There are no meaningless nodes in the tree.

In Figure 2.7-a, ***node 3***, ***node 4***, ***node 6*** and ***node 7*** represent terminals; ***node 1***, ***node 2*** and ***node 5*** represent functions of the program. In Figure 2.7-b, ***node 1*** is the only function and ***node 2*** is the only terminal of the program.

2.3.3. CREATION of the INITIAL POPULATION

Genetic programming starts with an initial population of randomly generated computer programs composed of functions and terminals appropriate to the problem domain. The creation of this initial random population is, in effect, a blind random search of the problem as represented by the computer programs (Koza 1994).

The process of creating a random tree structured computer program from the function and terminal sets can be seen as that of 'growing' a random tree of a specified initial depth. It is important to note that this is only the initial depth, as the shape and depth of the program will probably change once the adaptation process has begun.

The first step in the tree growing process is to choose a function at random from the function set to be the root of the tree. The number of parameters taken by this function is then extracted from the function set, and a corresponding number of branches are joined to the root function. For each of these newly created branches, a function is selected from the function set and attached to the branch. This process is repeated until the specified initial depth has been reached, at which point terminals are selected at random from the terminal set, and replaced at the end of every empty branch.

Koza added two features to this creation. Firstly, a generative method called "ramped half-and-half" is used by Koza to increase variety in the initial population. This method involves varying the initial depths of the population between 2 and the maximum specified initial depth. Secondly, before an individual is added to the population, the population is searched for a duplicate copy of the new individual. If one is found, the new program is discarded. This process is repeated until the whole population is full. This duplicate checking ensures 100% variety in the initial population (Koza 1992).

2.3.4. FITNESS

Each individual computer program in the population is measured in terms of how well it performs in the particular environment. The nature of the fitness measure varies with the problem. For some problems, the fitness of a computer program can be measured by the difference (i.e. error) between the result produced by the computer program and the correct result. The closer this is to zero, the better the computer program. This error is usually measured as a sum (or average) over a number of representative combinations of the inputs to the program (i.e. values of an independent variable). The fitness of an individual computer program in the population may be measured in terms of the sum, over the fitness cases, of the absolute value of the differences between the output produced by the program and the correct answer to the problem. (i.e. the Minkowski distance) or in terms of the square root of the sum of the squares (i.e. the Euclidean distance) (Koza 1992, Koza 1994, Langdon& Qureshi 1996). The correlation coefficient between the correct answer and the output produced by the program may be used in calculating the fitness (Koza 1994).

Unless the problem is so small and simple that it can easily be solved by blind random search, the computer programs in the initial population of the process will generally have exceedingly poor fitness. Nonetheless, some individuals in the population will turn out to be somewhat fitter than others (Koza 94). These differences in performance are then exploited.

2.3.6. REPRODUCTION

The reproduction operation involves selecting a computer program from the current population of programs on the basis of its fitness and allowing it to survive by copying it into the new population as is done in GAs.

2.3.7. CROSSOVER

A crossover operation is used to create new offspring computer programs from two parental programs selected on the basis of their fitness. The parental programs typically differ from one another in size and shape while the offspring are composed of subprograms of their parents. If two computer programs are effective in solving a given problem, then some of their parts probably have some merit. Crossing over these subprograms may yield a new program that is even more efficient at solving the problem (Koza 94).

Crossover operates on two parental computer programs selected with a probability based on fitness and produces two new offspring programs.

For example, after deciding to apply crossover according to the crossover probability, see Figure 2.8, parent-1 and parent-2 are selected via "roulette wheel" from the population of computer programs. Then, from random selection two nodes (i.e. node 5 in parent-1 and node 2 in parent-2) of two

subprograms cross over. Figure-3c shows one of the offspring. The other one is '$(x*x)/2$'.

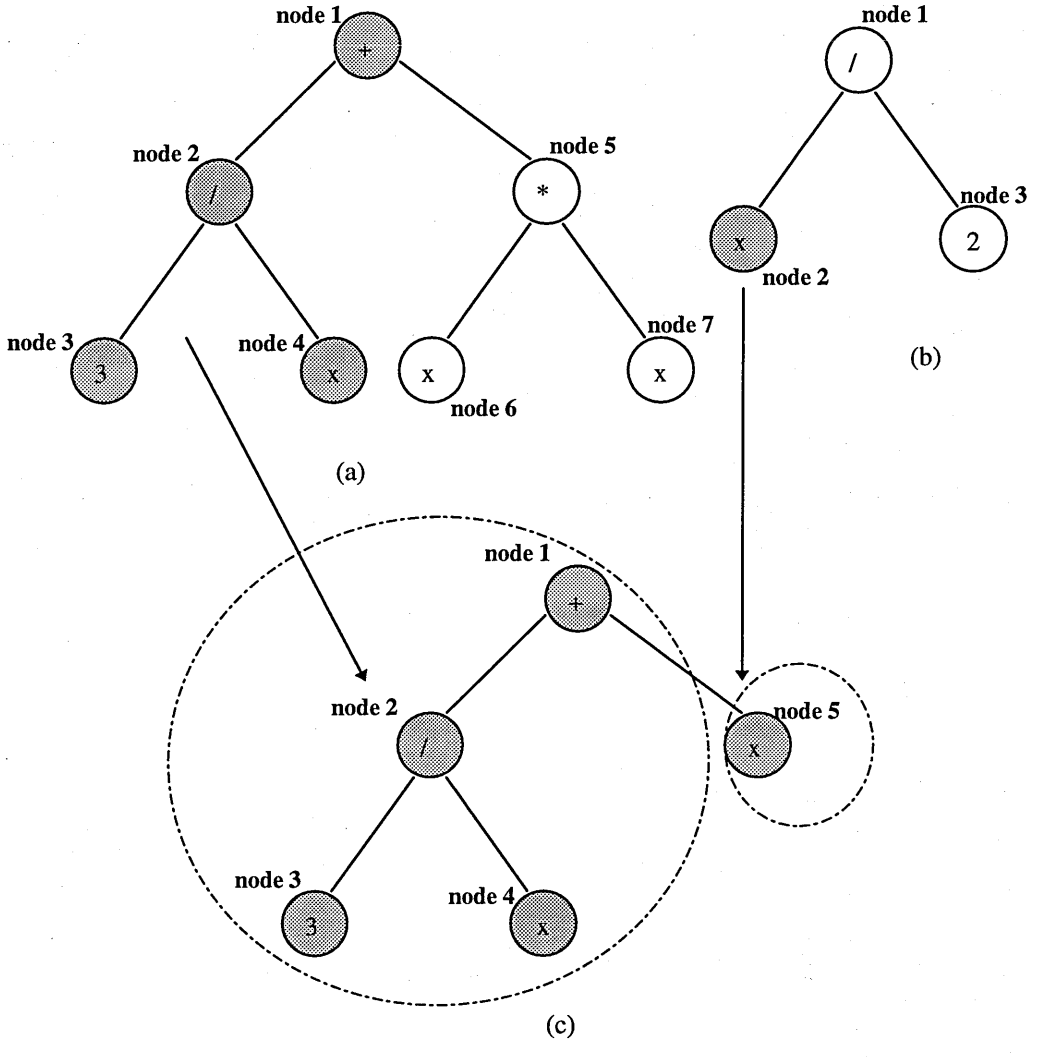


Figure 2.8. Two different individuals of different sizes in a population.

a. parent 1: $(3/x)+(x*x)$

b. parent 2: $x/2$

The crossover points:

node 5 in parent 1.

node 2 in parent 2.

c. offspring 1: $(3/x)+x$

2.3.8. MUTATION

The mutation operation replaces an internal node in a computer program by a randomly generated tree. The resulting computer program may add a new variety to the population of computer programs (Kinnear 1993 ,Koza 1994).

In Figure 2.9, after deciding to apply the mutation operation according to the mutation probability. Node 2 of the computer program in (a) was selected randomly as the point on which the mutation operator would be applied and so node 2 was replaced by the computer program (b) generated randomly.

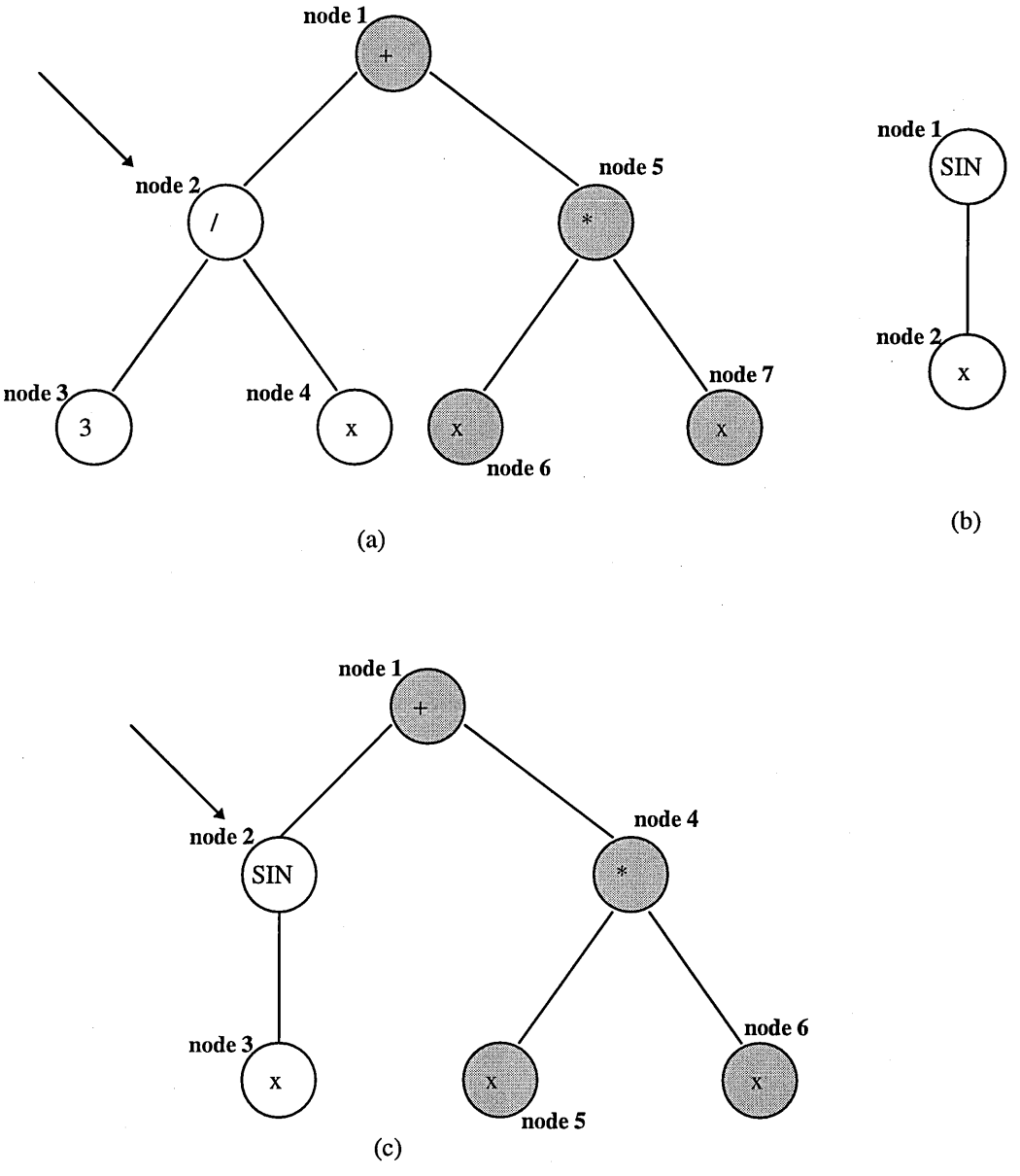


Figure 2.9. The Mutation Operation

a. parent : $(3/x)+(x*x)$

b. Randomly generated: $SIN(x)$
The mutation point:
the node 2 in parent (a).

c. offspring : $SIN(x)+(x*x)$

CHAPTER 3

COMPUTER IMPLEMENTATION of GAs&GP

3.1. INTRODUCTION

In this chapter, the data structures and the necessary algorithms to implement GAs and GP are explained. The Pascal programming language was chosen for the computer implementation. Two different systems designed for GAs and GP have been developed, using Borland Turbo Pascal version 6.0 on an IBM-PC compatible computer.

Although the systems have some similarities, there are differences in the way they implement the data structures, crossover, mutation and report the results. These are also explained in this chapter.

3.2. DATA STRUCTURES for GAs.

Genetic algorithms process populations of strings. Therefore, the primary data structure for the genetic algorithms is a string population. A population was constructed as a table where each row of the table contains properties of an individual. A schematic of a population is shown in Figure 3.1. The Pascal code of Figure 3.2 declares a population type corresponding to this model.

Referring to Figure 3.2, two constants are seen: the maximum population size, *maxpop*, and the maximum string length, *maxbyte*. These set upper bounds on the population size and the string length. Following the constant declarations, the population itself is declared, along with its components in the type block. The type population is an array of *individual_type* (indexed between 1 and *maxpop*). *Individual_type* is a record composed of an individual of *string_type*, a real variable called *fitness*, a real type variable called x, a real variable called *sofar*, and a real variable called *s_errors* . These represent the artificial individual, the string fitness value, the decoded parameter value x, the sum of

the fitness values of the individuals from the first one up to this one, and the sum of squared errors respectively.

Individual Number	*INDIVIDUALS*			
	STRING	*X*	*FITNESS*	*OTHER*
1	*01111*	*15*	*0.567*	...
2	*01001*	*9*	*0.234*	...
3				...
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
N	*00111*	*7*	*0.122*	

N:The number of individuals in the population

Figure 3.1. Schematic of a string population in GAs

```
const  maxpop       = 450;
       maxbyte      = 6;
type
       string_type          = array[1..maxbyte] of byte;
       decoding_type = array[1..maxbyte] of single;
       individual_type      = record
                                  individual    :string_type;
                                  x             :decoding_type;
                                  fitness       :single;
                                  sofar         :single;
                                  s_errors      :single;
                              end;
       population    = array[1..maxpop] of individual_type;
```

Figure 3.2. GAs data type declarations.

In the GAs, genetic operators are applied to an entire population at each generation, as shown in Figure 3.3. To implement this operation, two non-overlapping populations, the population of offspring (i.e. *newpop*) and the population of parents (i.e. *oldpop*), are utilised. With these two populations, it is a simple matter to create new offspring from the members of *oldpop* using the genetic operators, place those new individuals in *newpop*, and set *oldpop* to *newpop* when the genetic operators are applied to the whole population.

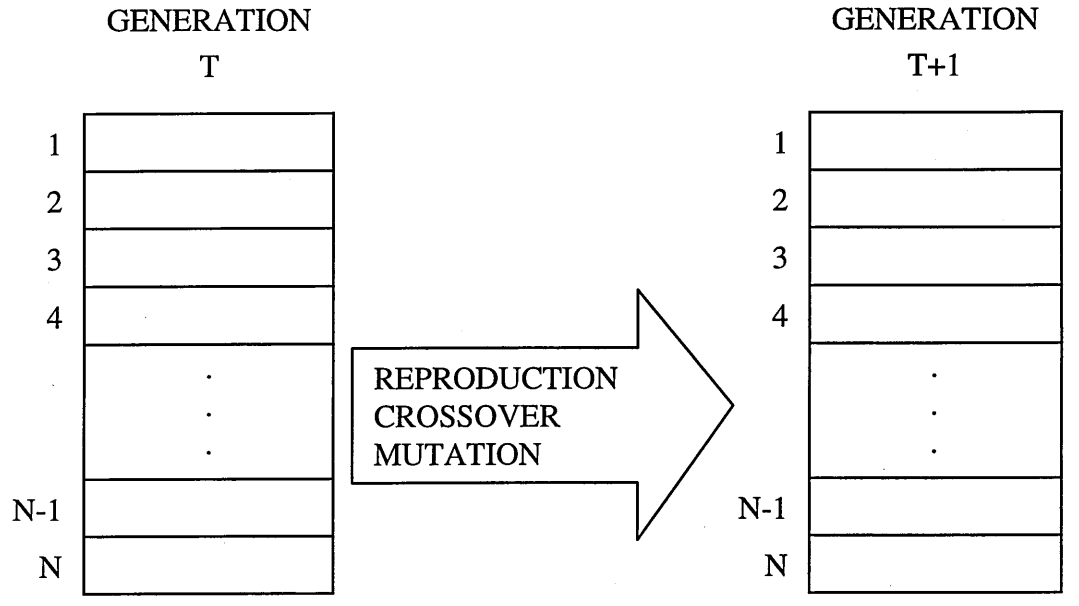


Figure 3.3. Schematic of non-overlapping populations used in GAs.

The criteria for the termination of the process is either when an individual string has the desired fitness value (e.g. *0.998*) or after a given number of generations (e.g. *maxgen is 50*).

3.3. REPRODUCTION, CROSSOVER, AND MUTATION for GAs

In the GAs, reproduction is implemented in the function *select* as a recursive search through a roulette wheel with slots weighted in proportion to string fitness values. In Figure 3.4, it is shown that *select* returns the population index value corresponding to the selected individual.

```
function select:integer;
{select one individual for mating via roulette wheel selection }
var rand:single; {random point on wheel }
   i:integer;        {population index}
   low,high:word;

procedure find(var low,high:word);
begin
   if (low+1=high) then exit;
   if pop[OLD_POP,low+(high-low) div 2].sofar>rand
         then high:=high-(high-low) div 2
         else low :=low +(high-low) div 2;
   find(low,high);
end;

begin
 i:=0; { Zero out counter }
 rand:=random * sumfitness; { Wheel point calc. uses random number [0,1] }
 if rand>pop[OLD_POP,1].sofar then begin
    low:=1;
    high:=popsize;
    find(low,high);
    select:=high;
 end
 else select:=1;
end;
```

Figure 3.4. Function *select* implements roulette wheel selection.

In this implementation, the real variable *rand* contains the location where the wheel has stopped after a random spin according to the computation:

*rand := random * sumfitness*

The real variable *sumfitness* represents the sum of the fitness values of whole population.

In the GAs, the crossover operator is implemented in a procedure called *crossover* (Figure 3.5). It takes two parent strings and generates two offspring strings. The crossover operator is performed according to the probability of crossover. The crossing site is selected randomly between 1 and *the length of the individual string-1*. Whilst the substrings are being exchanged on a bit-by-bit basis, the mutation operation takes place. For each bit being exchanged, a procedure called *mutation* is performed according to the probability of mutation (Figure 3.6). If the crossover operation is not performed, in order to perform the mutation operation, the crossing site is selected as the length of individual string. In the procedure *crossover*, this results in performing only the mutation, not the crossover.

```
procedure crossover(var parent1, parent2, child1, child2:string_type;
                    var lchrom, ncross, nmutation, jcross:integer;
                    var pcross, pmutation:single);
{ Cross 2 parent strings, place in 2 child strings }

var i:integer;
   byte_no,bit_no,kukla:byte;
begin
      i:=lchrom div 8;
        fillchar(child1,i,0);
        fillchar(child2,i,0);
        if flip(pcross) then begin      { Do crossover with p(cross) }
          jcross  := 1+random(lchrom-2); { Cross between 1 and l-1 }
          ncross := ncross + 1;         { Increment crossover counter }
        end
      else jcross  := lchrom; { Otherwise set cross site to force mutation }

       { 1st exchange, 1 to 1 and 2 to 2 }

for i:=1 to jcross do begin
    bit_no:=i mod 8;
    if bit_no=0 then bit_no:=8;
    byte_no:=i div 8;
    if bit_no<>8 then inc(byte_no);
    kukla:=parent1[byte_no] and two_1[bit_no];
    child1[byte_no]:=child1[byte_no] or mutation(kukla,bit_no);
    kukla:=parent2[byte_no] and two_1[bit_no];
    child2[byte_no]:=child2[byte_no] or mutation(kukla,bit_no);
  end;

  if jcross<>lchrom then
  for i:=jcross+1 to lchrom do begin
    bit_no:=i mod 8;
    if bit_no=0 then bit_no:=8;
    byte_no:=i div 8;
    if bit_no<>8 then inc(byte_no);
    kukla:=parent2[byte_no] and two_1[bit_no];
    child1[byte_no]:=child1[byte_no] or mutation(kukla,bit_no);
    kukla:=parent1[byte_no] and two_1[bit_no];
    child2[byte_no]:=child2[byte_no] or mutation(kukla,bit_no);
  end;
end;
```

Figure 3.5. Procedure *crossover* implements single-point crossover.

```
function mutation(_byte:byte;bit_no:byte):byte;
begin
   if flip(pmutation) then begin
        if _byte=0 then _byte:=two_1[bit_no] else _byte:=0;
    end;
   mutation:=_byte;
end;
```

Figure 3.6. Function *mutation* implements single-bit mutation.

3.4. CALCULATION of FITNESS

The calculation of the fitness of an individual is implemented in the procedure *fitness_function* (Figure 3.7). It uses the decoded parameter value x as an input, and generates two outputs, the fitness value and the sum of the squared values of the differences between the desired data and GAs' results.

```
procedure fitness_function(x:decoding_type; var diff,fitness:single);
{ Fitness function - f(x) }
var t:real;
   i:integer;
begin
   t:=0;
   for i:=1 to N do t:=sqr(desired^[i]-x[1])+t; {N is the number of  data in a set}
   fitness:=1/(1+t);
   diff:=t;
end;
```

Figure 3.7. Procedure *fitness* implements the calculation of fitness.

GAs uses the decoding routine shown in Figure 3.8, the procedure *decode*. In this procedure, a single individual string is decoded according to the researcher's application. The procedure returns the result by variable x.

```
procedure decode(individual:individual_type; var x:decoding_type);
begin
        x[1]:=0.25+individual[1]/8500;
end;
```

Figure 3.8. Procedure *decode* implements the decoding of an individual string.

3.5. OUTPUT of the GAs

The GAs generate a report file as an output which shows the parameters used for the experiment (i.e. population size, length of individual string, maximum generation number, probability of crossover and mutation), and lists the average fitness value, the best fitness value, the minimum sum value of the absolute error, and the value of the best individual at each generation (Figure 3.9).

The Desired Function =0.243

GAs Parameters

Pop.Size	*Gene Length*	*Max Gen.*	*Crossover p.*	*Mutation p.*
20	*8*	*10*	*0.500*	*0.006*

Generation	*Average Fitness*	*Min Differ.*	*1/(1+t) Fitness*	*A*
initial	*0.076*	*2.075*	*0.325*	*0.325*
1	*0.189*	*0.171*	*0.854*	*0.267*
2	*0.290*	*0.171*	*0.854*	*0.267*
3	*0.476*	*0.019*	*0.981*	*0.251*
4	*0.603*	*0.019*	*0.981*	*0.251*
5	*0.732*	*0.019*	*0.981*	*0.251*
6	*0.738*	*0.119*	*0.894*	*0.263*
7	*0.836*	*0.019*	*0.981*	*0.251*
8	*0.872*	*0.019*	*0.981*	*0.251*
9	*0.843*	*0.019*	*0.981*	*0.251*
10	*0.825*	*0.019*	*0.981*	*0.251*

Figure 3.9. An example output of the GAs.

3.6. DATA STRUCTURES for GENETIC PROGRAMMING.

In order to implement GP's individual program structure in tree form, pointers have been used. Here, the difference between GAs and GP is the definition of the individual member of a population. In GP, an interconnected collection of nodes in the memory makes an individual program. The type declaration of the node is shown in Figure 3.10. In this declaration, the record variable *code* designates whether the node is a function, a parameter of the function, or a constant. The record variable *value* designates the binary value of a constant, parameter number, or function number according to *code*. The record variables *back, left,* and *right* connect the nodes to each other. *Individual_type* is a record composed of a variable called *node_no*, a pointer type variable called *root*, and the type variables *fitness*, *diff*, and *sofar*. *Node_no* stores the number of nodes in the tree structure which was used to select a random point in a tree structured individual program for crossover and mutation operators. The type variable *root* points to the root node of the tree structure. The type variables *fitness*, *diff*, and *sofar* have the same function as in GAs.

```
type

_pointer      =^node_type;
node_type     = record
                    code   :byte;
                    value  :integer;
                    back   :_pointer;
                    left   :_pointer;
                    right  :_pointer;
                end;

individual_type= record
                    node_no     :word;
                    gene        :_pointer;
                    diff        :single;
                    fitness     :single;
                    sofar       :single;
            end;
```

Figure 3.10. The declaration of a node and an individual in GP.

In Figure 3.11, the list of the functions being used in trees is given by an array of strings called variable *symbols* while the number of parameters accepted by the functions is listed in the *param_number* variable. The other entries are the symbol c which represents constants, and x which represents variables. The constant *MAX_LEVEL_NO* stores the maximum level of a tree. This restriction is only used in the creation of the initial population whereas in the genetic process, trees may grow or shrink. The constant MAX_VAR_NO stores the maximum number of parameters necessary for the functions. The mutation operator generates a random subtree with the restriction *MUT_MAX_LEVEL_NO* as the maximum level number. To implement mutation and crossover operators, a recursive programming technique has been used (Baron 1968).

```
const

MAX_LEVEL_NO=5;
MUT_MAX_LEVEL_NO=2;
MAX_VAR_NO=1;

symbols:array[0..9] of string[5]=('c','x','+','-','*','/', 'sin','tan','sqr','sqrt');
param_number:array[0..9] of byte=(0,0,2,2,2,2,1,1,1,1);
```

Figure 3.11. Constant declarations for GP

In GP, decoding of the individual is different too, as GP decodes a tree structured individual using *Polish Notation* with a stack.

Additionally, GP prints out the best function in symbolic form at each generation (Figure 3.12).

```
original function : f(v)=0.756*sin((1.5*v+0.3)*2*pi)*cos(1.5*v)+noise

GAs Parameters
Pop.Size   Max Gen.   Crossover p.   Mutation P.
-----------------------------------------------------------
  500        100          0.750         0.003
-----------------------------------------------------------
Fitness:Correlation
 Gen. Av.Fit. Max Fit. Function
-------------------------------------------------------------------------
  0  0.113   0.691   sin(v*2*π+(-1.111))
  1  0.208   0.691   sin(v*2*π+(-1.111))
  2  0.223   0.835   v+(sin(v*2*π+(-0.988)))
  3  0.265   0.835   v+(sin(v*2*π+-0.988))
  4  0.297   0.824   v+sin(sin(v*2*π+(-1.111)))
  5  0.316   0.856   (1.346-v)*(sin(v*2*π+(-1.300)))
  6  0.321   0.829   (sin(v*2*π+(-0.666)))+v
  7  0.356   0.824   v+sin(sin(v*2*π+(-1.111)))
  8  0.385   0.853   sin(((-1.539)*v)*2*π+(-0.381))
  9  0.369   0.853   sin(((-1.539)*v)*2*π+(-0.381))
 10  0.387   0.898   sin((-1.191)*2*π+(sin(v*2*π+(-1.111))))
-------------------------------------------------------------------------
```

Figure 3.12. An example output of GP

CHAPTER 4

MODELLING & EXPERIMENTAL DESIGN

4.1. INTRODUCTION

This chapter introduces the modelling and experimental design of the research where GAs optimise the coefficients of predicted functions and GP approximates a function in symbolic form for data sets in hand. Two different types of GAs and GP were tested for symbolic regression. Each type was tried with different population sizes, different crossover and mutation probabilities, and different ways of calculating fitness. The data sets (i.e. parameters of functions and function outputs) were generated using a computer program. In chapter 6, the data sets which were taken from a real world problem, helicopter health and usage monitoring, have been used to approximate a function for the strain on a helicopter's main rotor.

4.2. MODELLING the SYSTEMS

For the experiments, two different types of GAs and GP were designed. Although the types of GAs only differ on implementing the crossover operator, the types of GP differ on implementing the structure of the individuals by applying crossover and mutation operators.

4.2.1. The FIRST MODEL for GENETIC ALGORITHMs (GAs-1)

In this model, the crossover operator of GAs is applied to an individual string at any point on a bit-by-bit basis as was described in Chapter 2 and the inputs are shown in Figure 4.1. Having given these inputs to the model, it tries to improve individuals' fitness applying genetic operators to the population of individuals. The input h_0 stands for the maximum fitness required by the researcher. If the model finds an individual with that fitness or bigger than

that, it stops. Input h_1 determines the calculation of coefficients from individual strings. Input h_2 determines the fitness function used. For example, it may be the absolute correlation coefficient between the desired data and the output of the model.

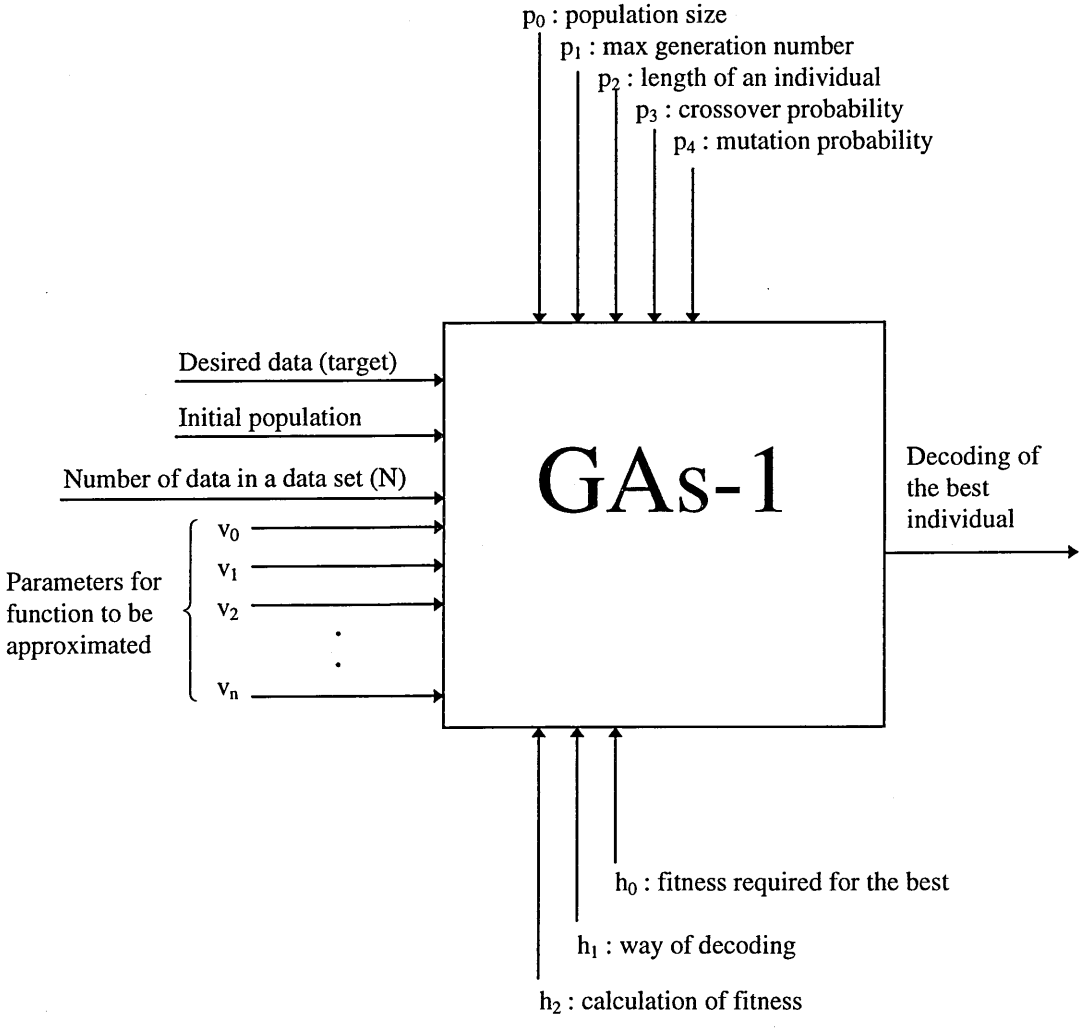


Figure 4.1. A model of the first system for GAs.

4.2.2. The SECOND MODEL for GENETIC ALGORITHMS (GAs-2)

This model differs from the first model on implementation of the crossover operator.

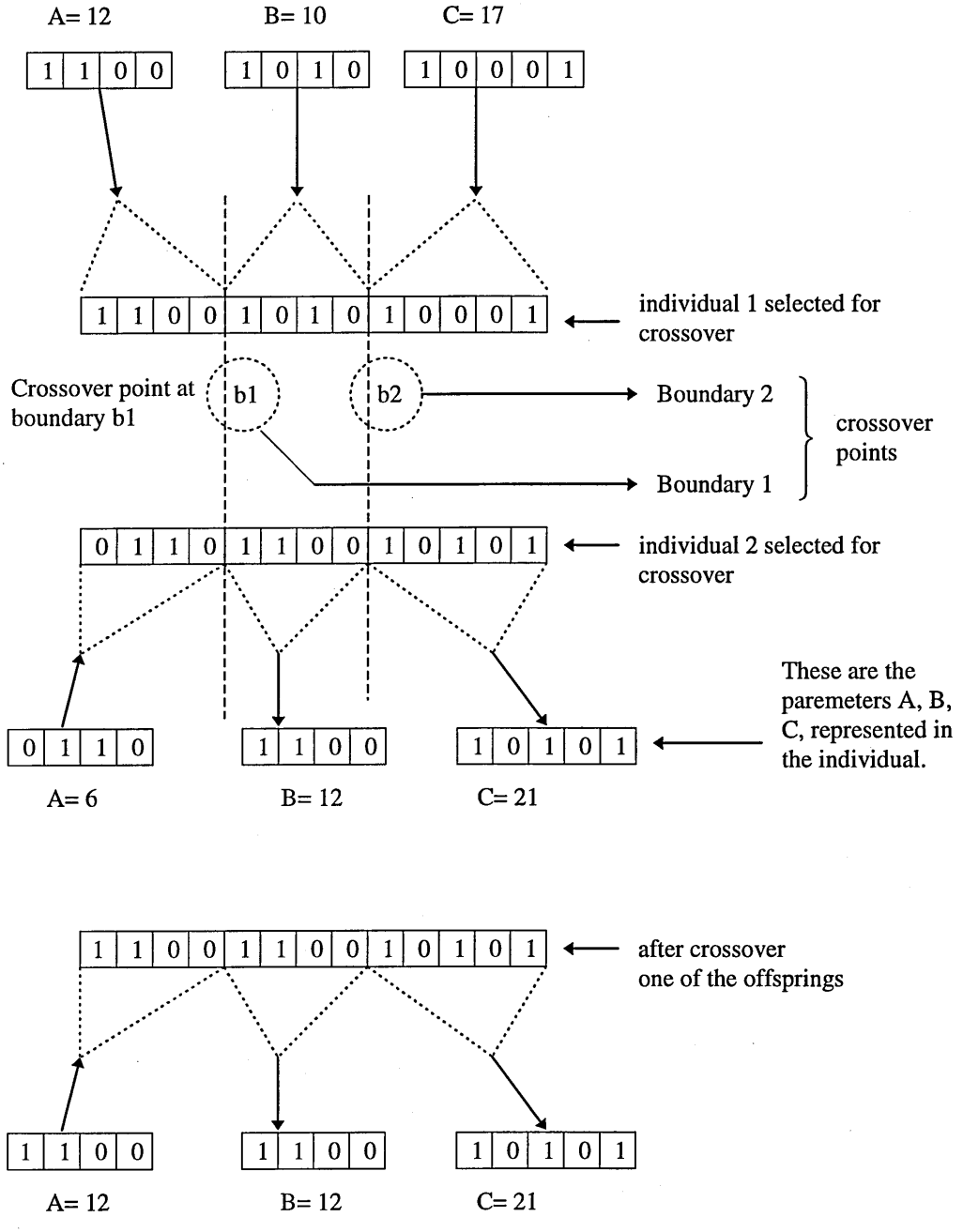


Figure 4.2. An example crossover for the second model of GAs.

In the second model of GAs, one of the boundaries between the parameters represented in the individual is selected as the crossover point (Figure 4.2). The crossover is not applied at any point within the parameters. The inputs and output of the model is designed as in the first model.

4.2.3. The FIRST MODEL for GENETIC PROGRAMMING (GP-1)

Instead of using the tree representation which is defined in the literature for GP, a fixed length binary string representation was used in this model. The aim is to apply the crossover and the mutation methods which are defined for GAs to GP. So that, GP can be tested using a different structure.

The binary string was decoded as a function written in Polish notation. Every sixteen bits represent a unit of Polish notation (Figure 4.3). In addition to the GAs' inputs, a list of basic functions is given as an input to the model. The input h_1 stands for the decoding of the constant unit (Figure 4.4). The output of the model is the best function printed out in symbolic form.

First eight bits were decoded as the type:

Whether the unit is

- the basic four arithmetic operator,
- a basic function given as an input,
- a constant, or
- a variable.

Second eight bits were decoded according to the type.

- one of the arithmetic operators,
- one of the basic functions,
- the value of constant, or
- the input variable sequence number.

Figure 4.3. The decoding of a unit in an individual

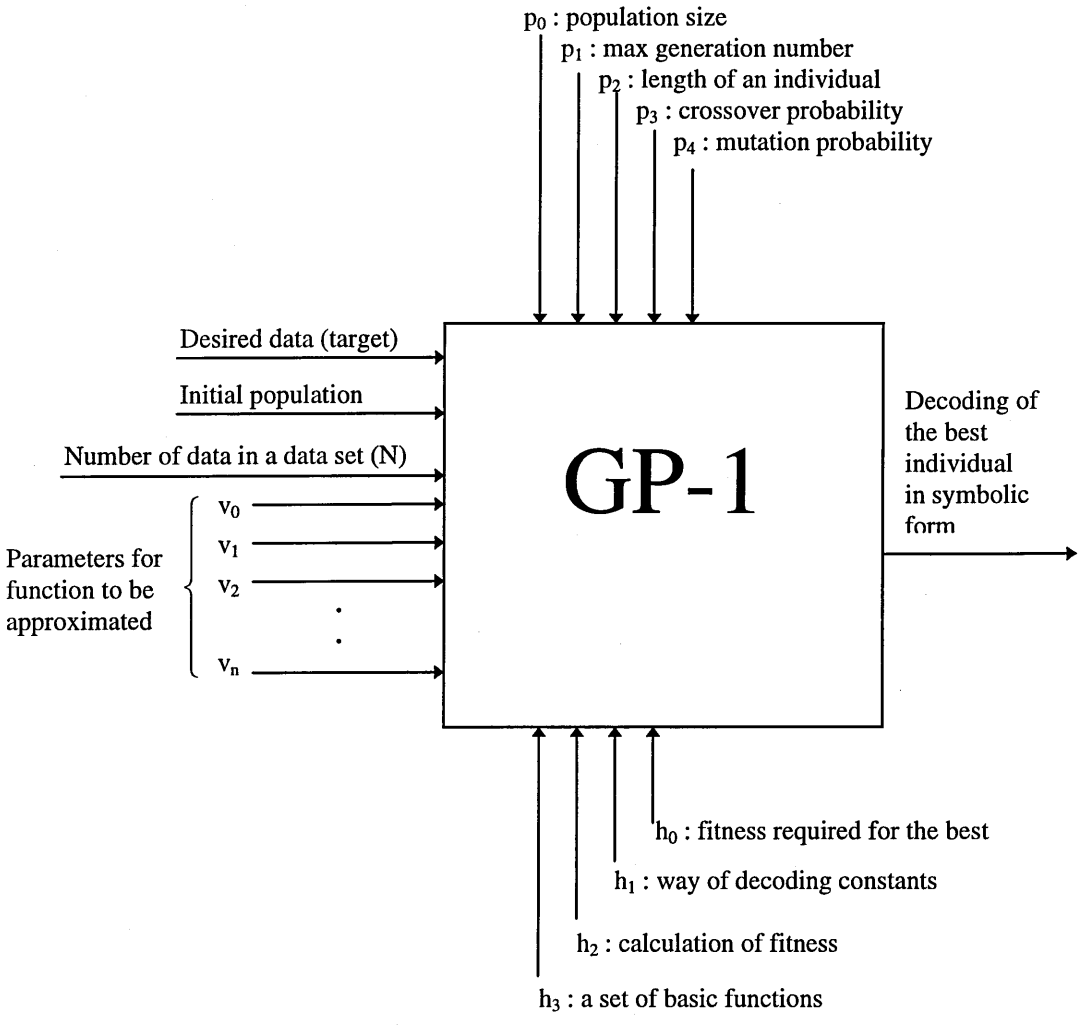
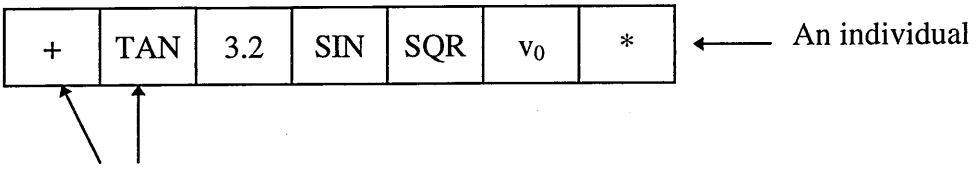


Figure 4.4. A model of the first system for GP.

An individual is made up from the units. Assuming that the individual is written in Polish notation, it makes a lot of units useless in the individual (Figure 4.5).

The crossover and mutation operators are applied as in GAs-2. Therefore, the crossover points are boundaries of the units.



That individual represents the function:

$$f(v_0)=v_0*SIN^2(3.2)$$

Figure 4.5. An individual in GP-1.

4.2.4. The SECOND MODEL for GENETIC PROGRAMMING (GP-2)

This model is designed to apply genetic programming as defined in the literature. Inputs and output of the system were designed as in Figure 4.6. All these inputs were explained in Chapter 2 and Chapter 3. In this representation, there is no meaningless node in a tree structured individual.

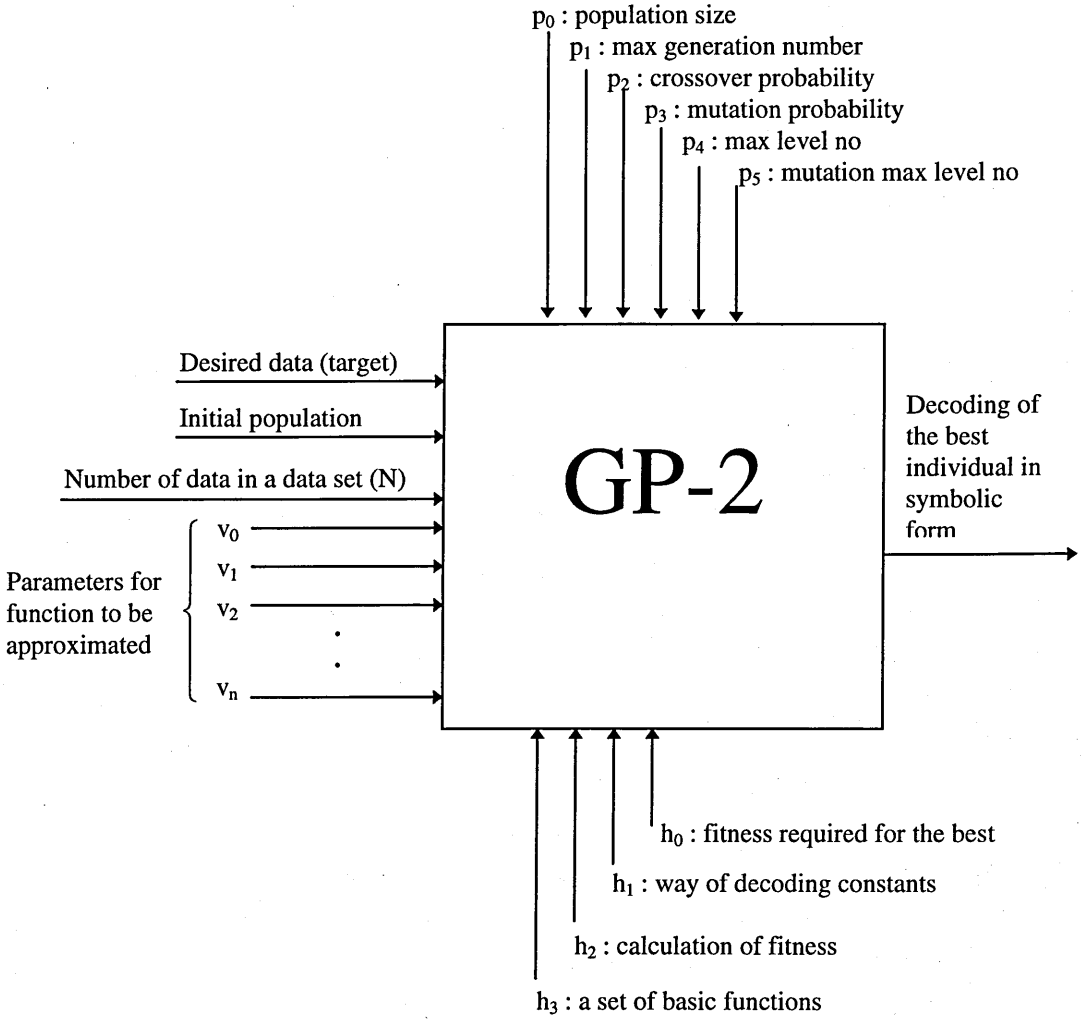


Figure 4.6. A model of the second system for GP.

4.3. EXPERIMENTAL DESIGN

To explore the models, a number of experiments were undertaken using different inputs. These designs were produced in order to see whether the models work properly, and also to seek the right genetic parameters.

4.3.1. The DESIGN for EXPERIMENTS USING GAs-1 & GAs-2

For the experiments undertaken using the model Gas-1, the following inputs were used.

• The desired data sets:

Three different types of function data sets were generated as the desired data sets. These are:

1. f = 0.243 (i.e. constant),
2. f(v) = 0.45* v +0.85 (i.e. a linear function),
3. f(v) = 0.43*SIN((2* v +0.30)*2*π) (i.e. a sine function)
4. f(v)=(0.756+ **n**)*Sin(((1.5+**n**)*v+(0.3+**n**))*2*π)*Cos((1.5+**n**)*v)

   n:noise

• Initial population:

For the first function five initial populations, for the second and third function ten initial populations were generated randomly. GAs-1 worked on each population using different population sizes, crossover probabilities, and mutation probabilities.

• The amount of data in a set:

There are 152 data items in a set.

• The parameter inputs for the function to be approximated:

One input parameter was used for the function 2 and 3. The parameter v was kept between 0 and 1.

• The fitness value required for the best individual is 0.998.

• The decoding of an individual:

The following variables A, B, and C were used in the fitness calculation.

→ For the first function, an individual was decoded using the formula:

bv: Decimal value of the individual

$$A = \frac{bv}{255}$$

The coefficient 'A' is expected to be in the range 0-1:

→ For the second, the formula

bv_1: Decimal value of the first 8 bits of the individual

bv_2: Decimal value of the second 8 bits of the individual

$$A = \frac{bv_1}{255} \qquad B = \frac{bv_2}{255}$$

were used for the decoding.

The coefficients 'A' and 'B' are expected to be in these ranges:

A	B
0-1	0-1

→ For the third function, the formula

bv_1: Decimal value of the first 8 bits of the individual

bv_2: Decimal value of the second 8 bits of the individual

bv_3: Decimal value of the third 8 bits of the individual

$$A = \frac{bv_1}{255} \qquad B = \frac{bv_2}{120} \qquad C = \frac{bv_3}{255}$$

were used.

The coefficients 'A', 'B' and 'C' are expected to be in these ranges:

A	B	C
0-1	0-2.125	0-1

→ For the fourth function, the formula

bv_1: Decimal value of the first 8 bits of the individual

bv_2: Decimal value of the second 8 bits of the individual

bv_3: Decimal value of the third 8 bits of the individual

bv_4: Decimal value of the third 8 bits of the individual

$$A = \frac{bv_1}{255} \qquad B = \frac{bv_2}{120} \qquad C = \frac{bv_3}{120} \qquad D = \frac{bv_4}{255}$$

were used.

The coefficients '*A*', '*B*', '*C*' and '*D*' are expected to be in these ranges:

A	*B*	*C*	*D*
0-1	0-2.125	0-2.125	0-1

• The fitness was calculated using the square of the absolute error between the GAs' output and the desired data as in the formula:

$$\text{fitness} = \frac{1}{1+E}$$

E is calculated using following formula:

$$E = \sum_{i=1}^{N} (\text{desired}_i - y_i)^2 \quad \text{or} \quad E = \sum_{i=1}^{N} |\text{desired}_i - y_i|$$

while 'y_i' is

→ For the first function:

$$y_i = A$$

→ For the second one:

$$y_i = A * v_i + B$$

→ For the third one:

$$y_i = A * \sin((B * v_i + C) * 2 * \pi)$$

→ For the fourth one:

$$y_i = A * \sin((B * v_i + C) * 2 * \pi) * \cos(D * v_i)$$

The absolute correlation coefficient between GAs' output and the desired data was used as the fitness function in order to see its effects. The correlation coefficient was calculated using the formula:

$$r = \frac{N*\sum_{i=1}^{N}(Di*Oi) - (\sum_{i=1}^{N}Di)*(\sum_{i=1}^{N}Oi)}{\sqrt{\left[N*\sum_{i=1}^{N}Di^2 - (\sum_{i=1}^{N}Di)^2\right]*\left[N*\sum_{i=1}^{N}Oi^2 - (\sum_{i=1}^{N}Oi)^2\right]}}$$

D : Desired data

O : Output of GAs-1

r : Correlation coefficient

Having completed that part, the GAs-1 worked on one of the initial populations to assess the improvements of the population in terms of the fitness generation by generation for each type of the desired data function sets.

• Population size:

This was kept in the range [2-200].

• The genetic parameters:

→ Maximum generation number was kept in the range 100-4000.

→ The crossover probability was kept in the range [0.2-1.0].

→ The mutation probability was kept in the range [0.001-0.009].

• The length of an individual:

This was 8 bits for the first function, 16 for the second, 24 for the third, 32 for the fourth.

4.3.2. The DESIGN for EXPERIMENTS USING GP-1

In these experiments, the objective was to test the fixed length binary string representation for GP and to seek the effects of different lengths of an individual. The model GP-1 was tested using four desired data sets of different difficulties.

• Desired data set

The three different sets that were generated for the model GAs-1 were used (i.e. constant, linear, and sinusoidal functions).

• Initial population:

The initial populations were generated randomly.

• The amount of data in a set:

There are 152 data items in a set.

• Parameters for the function to be approximated:

One variable was used as the input variable of the desired data. This is the one used for GAs-1.

• The following set of basic functions was used:

→ +,-,*,/,sin(x,y,z), sin(x), tan(x),sqr(x), and sqrt($|x|$).

sin(x,y,z)=x*sin(y*2*π+z)

• The fitness value required for the best individual was 0.950.

• Decoding constants:

In this model constants were represented using an 8-bit structure. For the decoding, the following formula was used:

bv:Binary value of the constant.

vc: Value of constant.

$$vc = \frac{bv}{255}$$

• Calculation of fitness:

The sum of the squared differences and the correlation coefficient between the desired data and the output of GP-1 were used in the fitness calculation as in GAs'.

• Population size:

This was kept in the range [20-200].

• Genetic parameters:

→ Maximum generation number was kept in the range [50-4000].

→ The length of an individual was kept in the range [16-640 bits]

→ Crossover probability was kept at 0.750.

→ Mutation probability was kept at 0.003.

4.3.3. The DESIGN for EXPERIMENTS USING GP-2

In these experiments, the model GP-2 was tried using the following set-up:

• Desired data set

The three different sets generated for the model GAs-1 were used

• Initial population:

The initial populations were generated randomly.

• The number of data in a set:

There are 152 (i.e. N) data in a set.

• Parameters for function to be approximated:

One variable was used as the input variable of the desired data. This is the one used for GAs-1.

• The following basic functions were used:

→ +,-,*,/,sin(x,y), sin(x), tan(x),sqr(x), and sqrt($|x|$).

sin(x,y)=sin(x*2*π+y)

• The fitness value required for the best individual was 0.950.

• Decoding constants:

In this model constants were represented using a 16-bit structure. For the decoding, the following formula was used:

iv: Integer value of the 16 bits.

iv is an integer number between -32768 to 32767.

vc: Value of constant.

$$vc = \frac{iv}{20000}$$

vc is a real number between -1.638 and 1.638.

• Calculation of fitness:

The sum of the squared differences and the correlation coefficient between the desired data and the output of GP-2 was used in the fitness calculation as was in GAs.

• Population size:

This was kept in the range [20-500].:

• Genetic parameters:

→ Maximum generation number was kept at 50.

→ Crossover probability was kept in the range [01-1.0].

→ Mutation probability was kept in the range [0.001-0.003].

• The maximum level of an individual tree was kept at 3.

• The maximum level of a subtree generated for mutation was kept at 2.

CHAPTER 5

DISCUSSION on EXPERIMENTAL RESULTS

5.1. INTRODUCTION

This chapter discusses the results of the experiments which were designed to explore genetic algorithms and genetic programming towards symbolic regression using different structures. The first two models (i.e. GAs-1 and GAs-2) were tested to optimise coefficients of predicted functions. The other two models (i.e. GP-1 and GP-2) approximate a function in symbolic form from a set of basic functions and an independent parameter. The target was to test the models and to seek the best genetic parameters and inputs associated with the structures of the models.

5.2. The DATA SETS USED in the EXPERIMENTS

Three different types of function were used in this part of the experiment. No noise was added to the data generated using these functions. In chapter 6, a data set which is taken from an actual problem will be used. To prepare a good model for this problem, the genetic structures need to be explored. As the independent parameter for the functions, a set of 152 real values between 0 and 1 was generated. This parameter is called v (Figure 5.1).

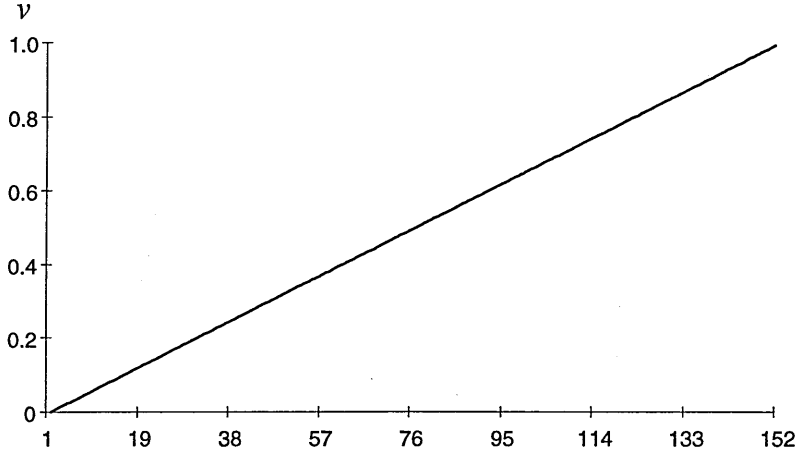


Figure 5.1. The graph of the parameter v

Using the values of v, three functions were generated called *constant*, *linear* and *sinusoidal*. The graphs of these are shown in Figure 5.2, Figure 5.3, and Figure 5.4.

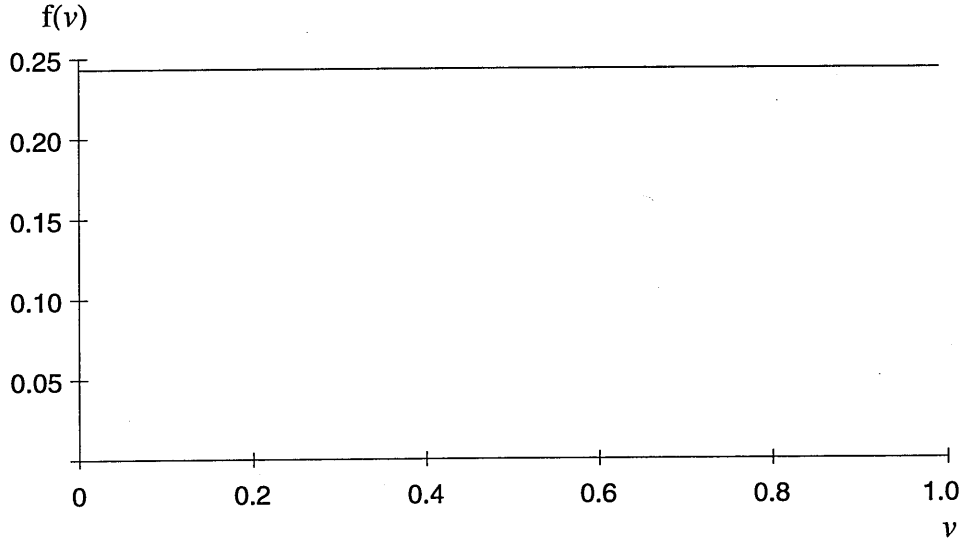


Figure 5.2. The graph of the constant function f(v)=0.243

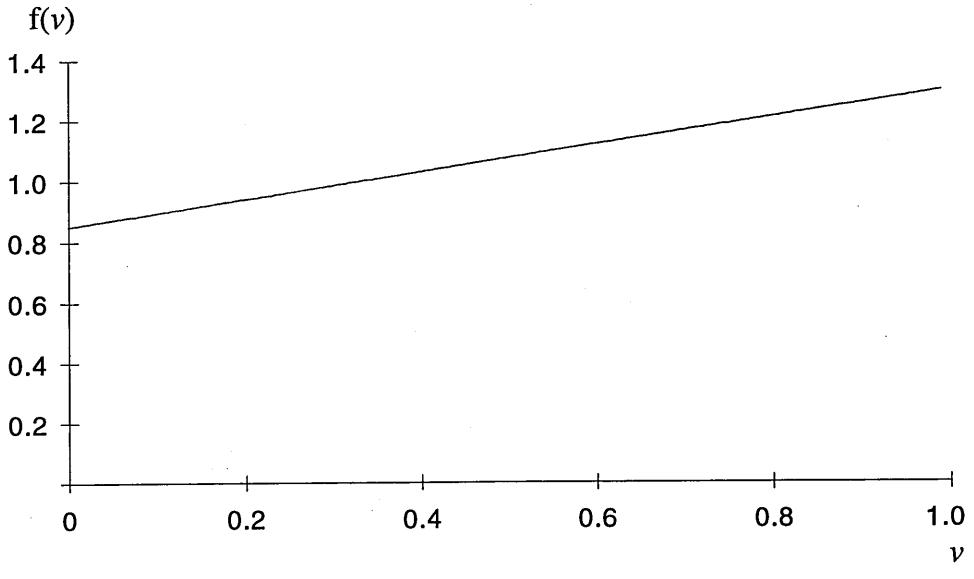


Figure 5.3. The graph of the linear function f(v)=0.45*v+0.85

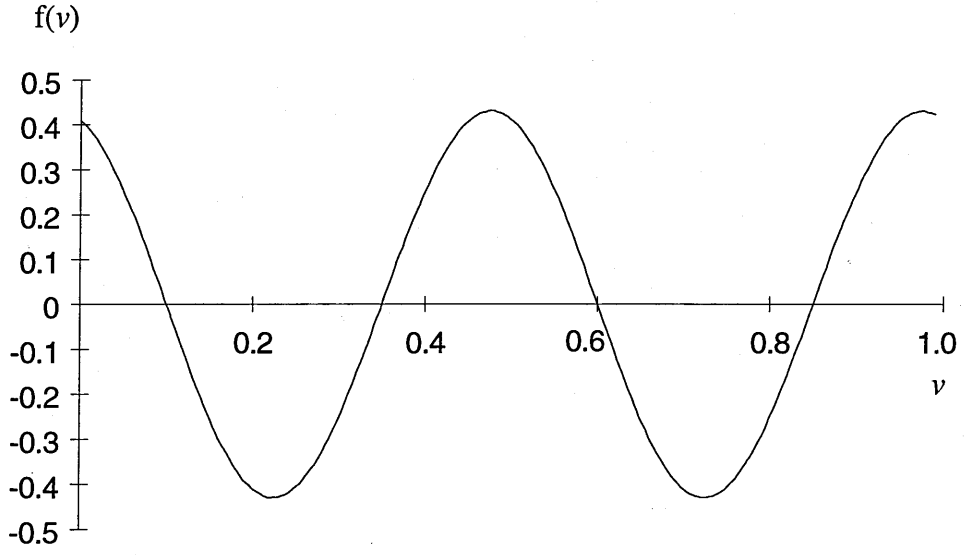


Figure 5.4. The graph of the sinusoidal function f(v)=0.43*sin(4*v*π+1.88)

As a final test the following noisy function was used:

n:noise

f(v)= (A+ **n**)*Sin(((B+ **n**)*v+(C+ **n**))*2*π)*Cos((D+ **n**)*v)

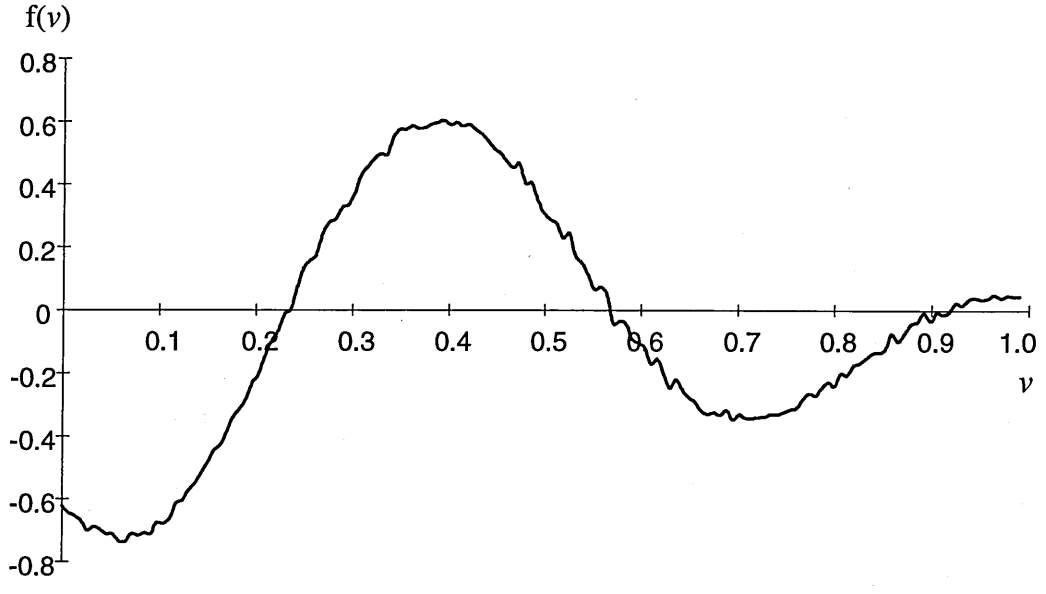


Figure 5.5. The graph of the function

f(v)= (A+ **n**)*Sin(((B+ **n**)*v+(C+ **n**))*2*π)*Cos((D+ **n**)*v)

5.2. SOLUTION SPACE

The representations and the decoding methods used for optimising the coefficients of the three functions were explained in chapter 4.

For the first function, the representation was an 8-bit binary string. There are 256 different solutions to the problem because of the decoding method, and these make up the solution space. One of these solutions may fit the problem. That means the representation and the decoding of an individual determine the solution space.

Function no	Number of solutions
1. Constant	2^8=256
2. Linear	2^{16}=65,536
3. Sine	2^{24}=16,777,216
4. Noisy	2^{32}=4,294,967,296

Table 5.1. The number of different solutions for the functions

Table 5.1 shows the number of different combinations of coefficients which make up the solution spaces in the GAs-1 and the GAs-2.

In the GP-1, the solution space is determined by the set of basic functions, the number of independent parameters, the decoding method for constants, and the length of an individual string. As the length of an individual increases, the number of possible solutions increases. For example, if the target is the function *constant* and the length of the individual is 16, the GP-1 can only find

the constant '0.243' as a perfect solution. If the length of an individual is set to 48, '0.1647+0.078' can be one of the best solutions as well as the constant '0.243'.

In the GP-2, the solution space is determined by the set of basic functions, the number of independent parameters, the decoding method for constants. Because the trees may grow or shrink in the genetic process, the size of the solution space changes. The experiments showed that the GP-2 can find many different solutions to problems as in GP-1.

5.3. The IMPORTANCE of the FITNESS FUNCTION

The selection of the fitness calculation method plays an important role in genetic algorithms and genetic programming. It is important that the error-driven calculation fits both the desired data and the output of the models. The correlation coefficient between the desired data and the output of a model may be used in a fitness calculation. This may help the models to find the functions which look like the desired data's function. A combination of both methods may be used. For example, the experiment 5.8.8 shows that this strategy drives genetic algorithms better than previous strategies.

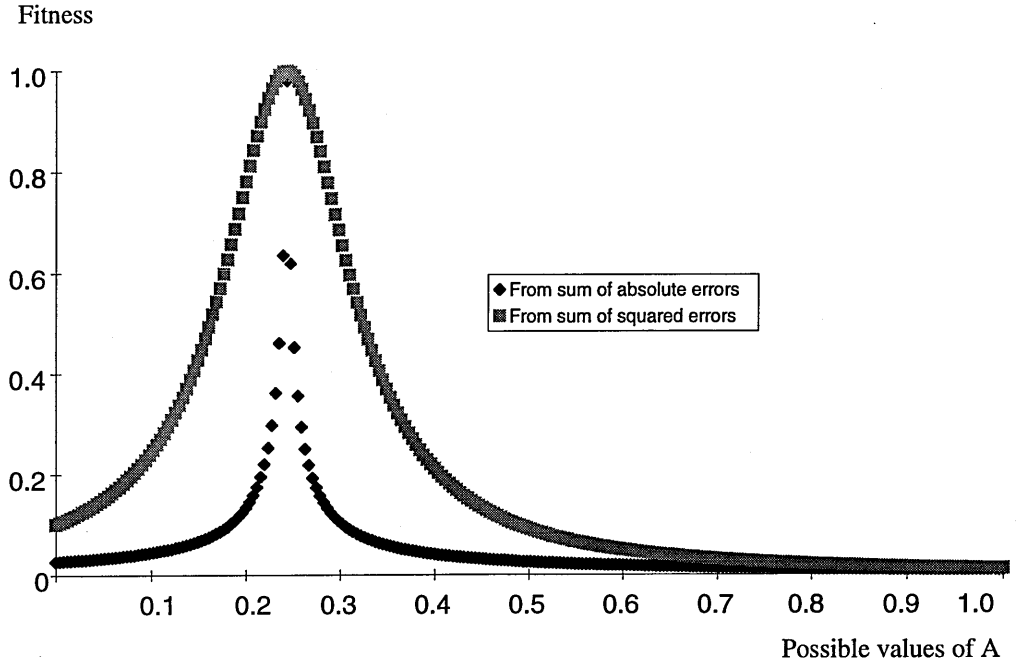


Figure 5.6. Fitness values vs possible values of A for the function *constant*.

For the first desired data set (i.e. constant), as shown in Figure 5.6, using the sum of the absolute errors in the fitness calculation dramatically increases the fitness around the solution. If an individual's fitness increases gradually around the best one, this may help genetic algorithms and genetic programming to find the best. The use of the sum of the squared errors helps the models to search for the best solution in a better mapped fitness spectrum.

For the second desired data set (i.e. linear), the following figures show that the use of the sum of squared errors in the fitness calculation maps the solution space better.

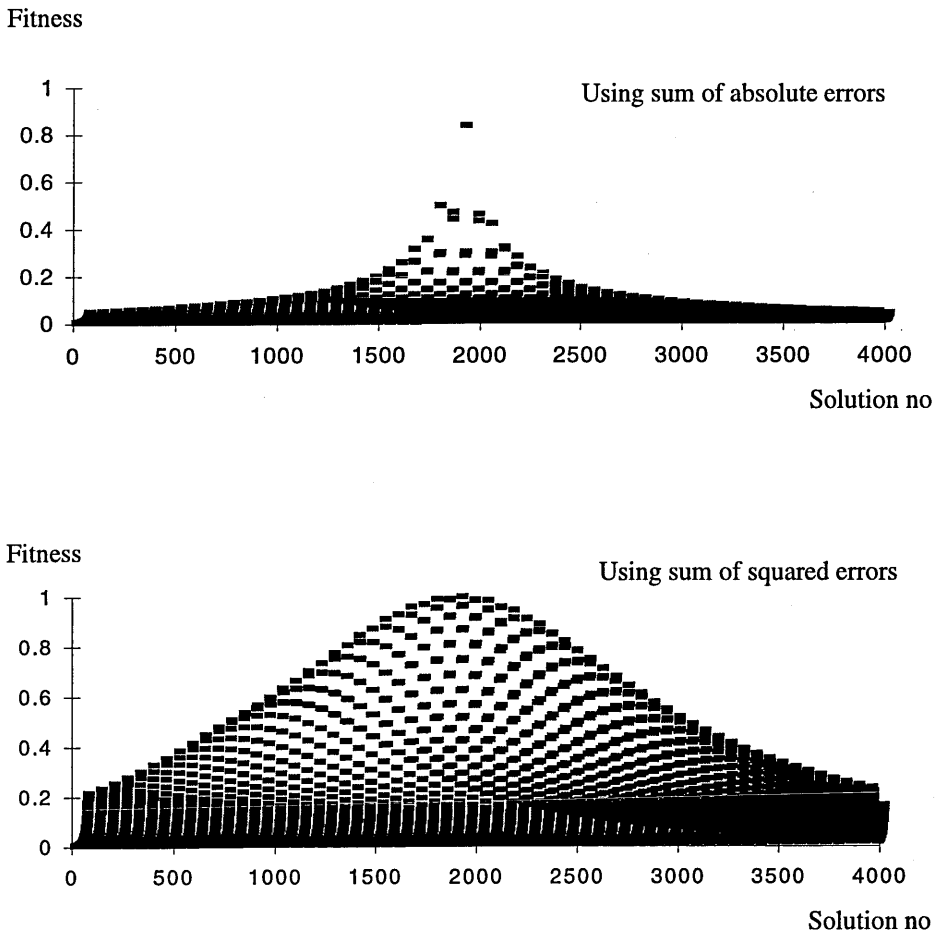


Figure 5.8. The fitness spectrum of the problem.

If the absolute error between the desired data and a model's output is bigger than 1, the sum of the squared errors may not help the models.

In some of the experiments, the absolute value of the correlation coefficient was used as the fitness function. Although there is only one perfect solution using the sum of the squared errors, there may be more than one perfect solution using the correlation coefficient in a fitness calculation.

5.4. The EFFECT of SELECTING a POPULATION SIZE

The four models tested used different population sizes. Although the minimum population size must be at least 2 in order to apply genetic operators, using bigger population sizes may result in getting better outputs from the models. Figure 5.9 shows the results of a set of experiments undertaken using the model GAs-1. In these experiments ten different initial populations were generated randomly. The population sizes and the other genetic parameters were changed as shown in the figure. These experiments suggest that using bigger population sizes may help the models to find the best individual in fewer generations. The population size must be selected according to the solution space. Otherwise it may take many generations to find a good solution. The bigger solution space requires bigger population sizes. If a solution space size is not known, as in the model GP-1, or the space changes during a genetic process as in the model GP-2 , the population size must be as big as possible.

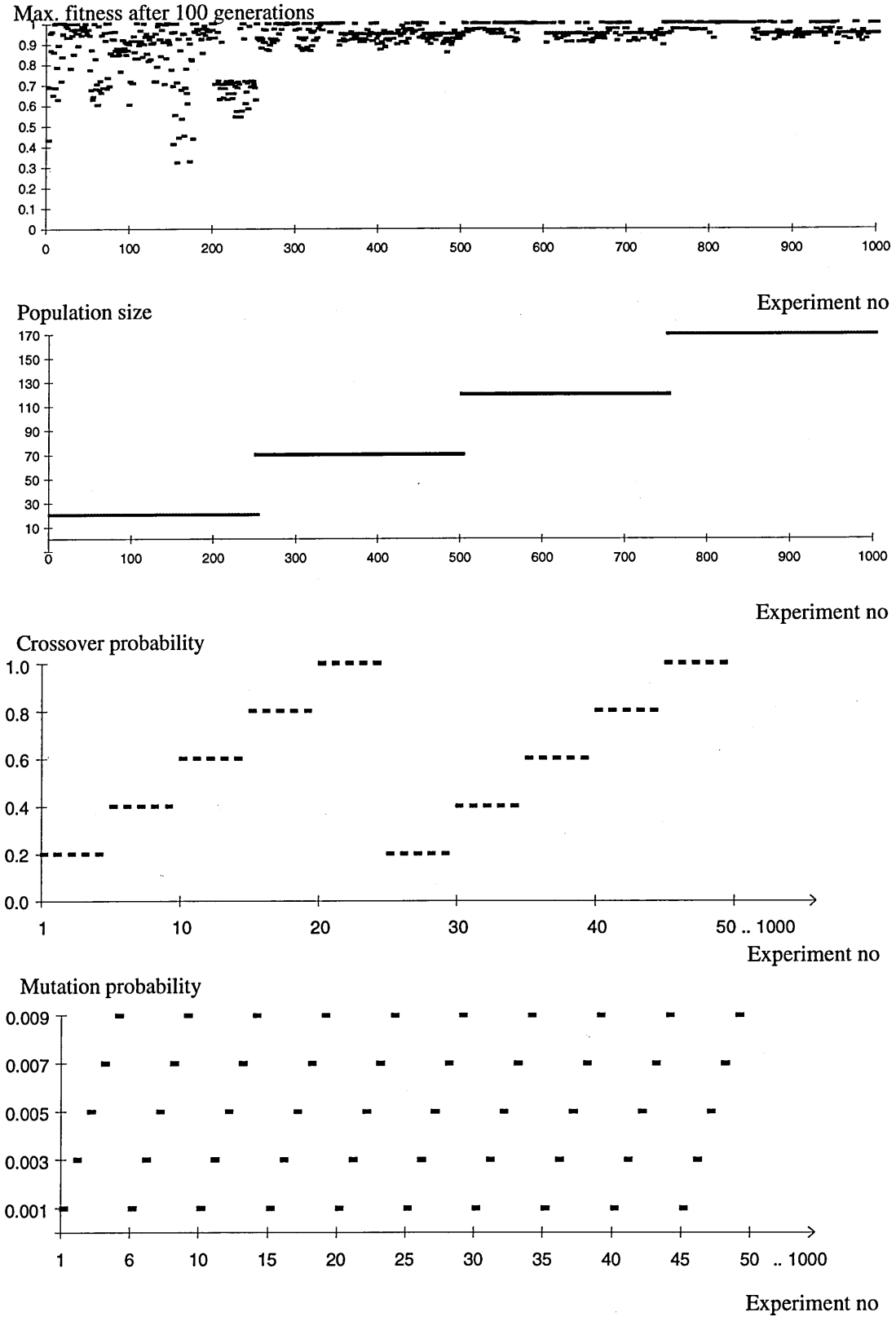


Figure 5.9. The effect of the population size on the fitness while the other parameters are changing for the function *linear* using GAs-1.

5.5. CROSSOVER PROBABILITY

Although crossover probability may be any real number between 0 and 1, some members of an old population should be copied without applying the crossover operator. If it is 1, it means that a model always applies the crossover. In that case the model may be turned into a random search mechanism. The best crossover probability should be selected according to the solution space. If it is a difficult one, the crossover probability must be kept small. In a genetic process, it is important to keep the average fitness of the population increasing. This may be done by allowing some members of an old population to be copied into a new population and then applying the crossover to the others.

For the models GAs-1 and GAs-2, if the crossover rate is selected as 0.750, the models' outputs give what is expected from a genetic model. The rate allows 25% of the existing generation to be copied into the next generation without applying crossover operation.

For the model GP-1 and GP-2, if the fitness function was calculated using the sum of squared errors, the crossover probability must be kept as low as 0.100. Otherwise the models turn into a random search mechanism, and the average fitness does not increase, but decreases. The reason is that they try to approximate a function which may resemble the desired one. They may find many functions which have a good correlation coefficient but a bad mean absolute error value. Using correlation coefficient in these models adds some more good solutions to a solution space. In that case the crossover probability may be 0.750, but this may vary from one problem to another.

Selecting a crossover probability for experiments must be done after testing a problem domain using different probabilities. For final experiments, a crossover probability which increases average fitness must be selected.

5.6. MUTATION PROBABILITY

A mutation operator may add new features to a population. The effect of mutation probability depends on the representation and the population size. The operator must not decrease average fitness. It may oscillate around a level, but must not go down so far. Therefore, the mutation probability must be kept low.

For the models GAs-1, GAs-2, and GP-1, the number of the bits on which the mutation may be applied in one generation is calculated using the following formula:

N_b : number of bits on which mutation may be applied

l : length of an individual.

N : population size

P_m : mutation probability

$$N_b = P_m * l * N$$

For example:

l = 24

N = 150

$P_m = 0.003$

Using these parameters

$$\Rightarrow N_b = 0.003 * 24 * 150 = 10.8$$

$$\approx 10$$

The mutation may be applied on ten bits of the population.

For the model GP-2, because the mutation operator changes the number of nodes in an individual, the mutation may change the individual drastically.

5.7. TESTING the MODEL GAs-1

Experiment 5.7.1:

Desired function	f(v)=0.243
Population size	2
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	1000

Fitness function:

$$f = \frac{1}{1 + \sum error_i^2}$$

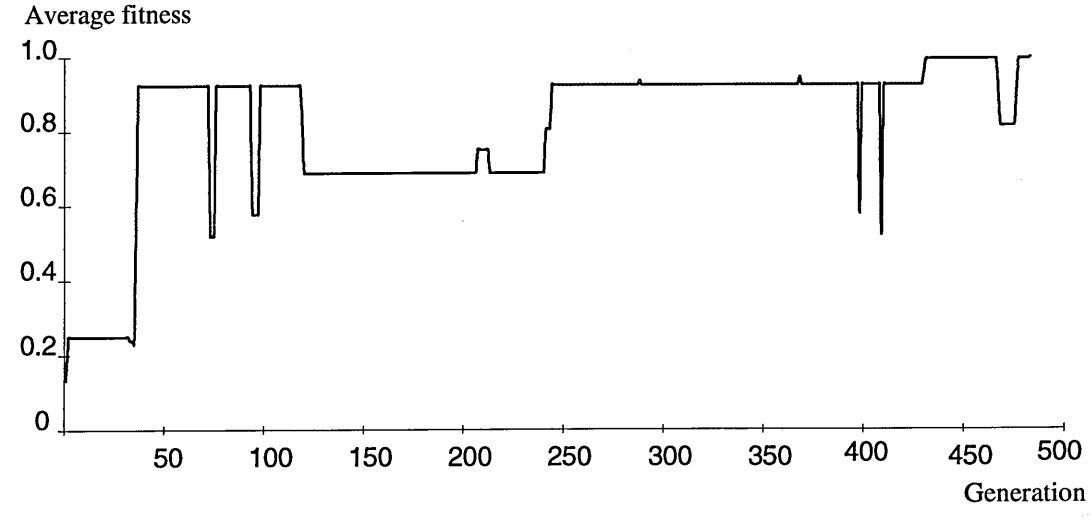


Figure 5.10. Average fitness vs. generation number.

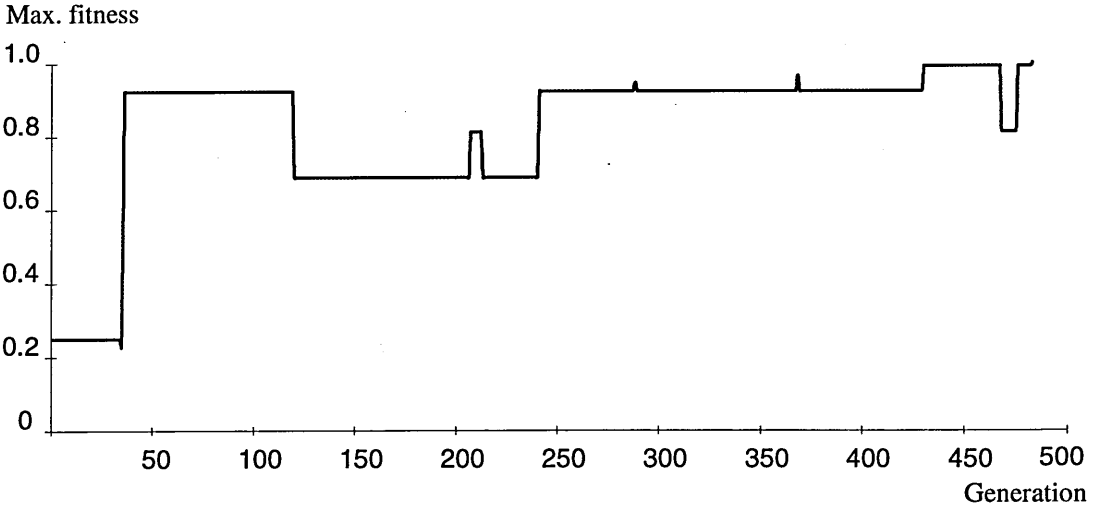


Figure 5.11. Maximum fitness vs. generation number.

Experiment 5.7.2:

Desired function	f(v)=0.243
Population size	10
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	1000

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

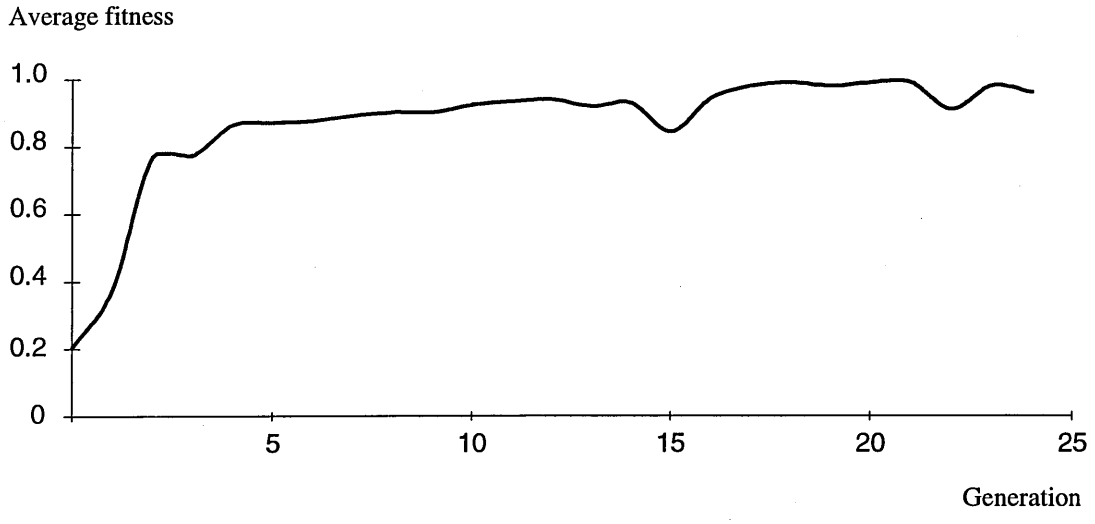


Figure 5.12. Average fitness vs. generation number.

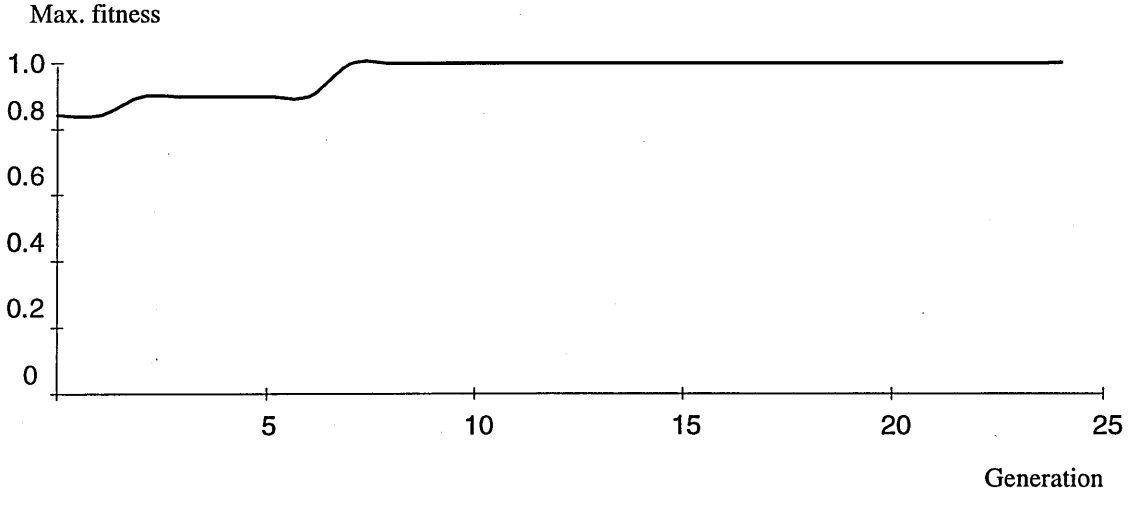


Figure 5.13. Maximum fitness vs. generation number.

Experiment 5.7.3:

Desired Function	f(v)=0.45*v+0.85
Population size	10
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	4000

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

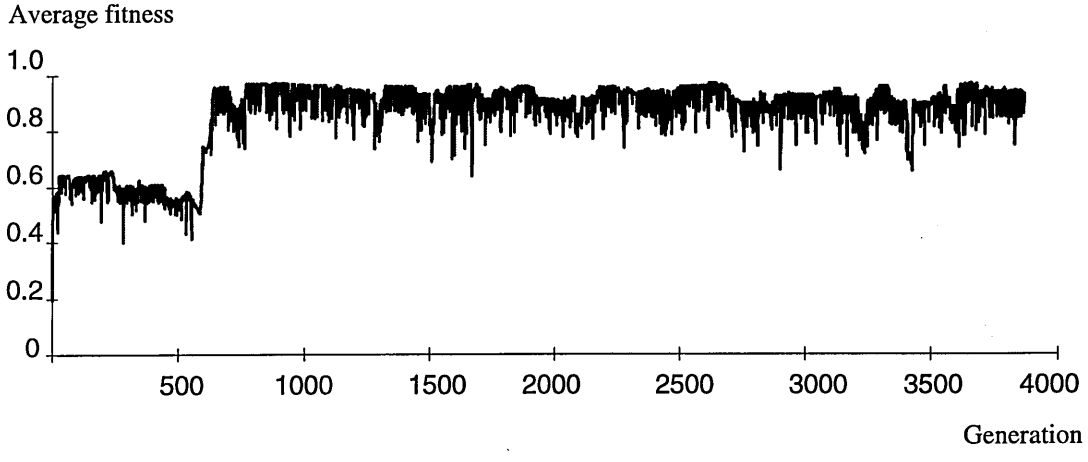


Figure 5.14. Average fitness vs. generation number.

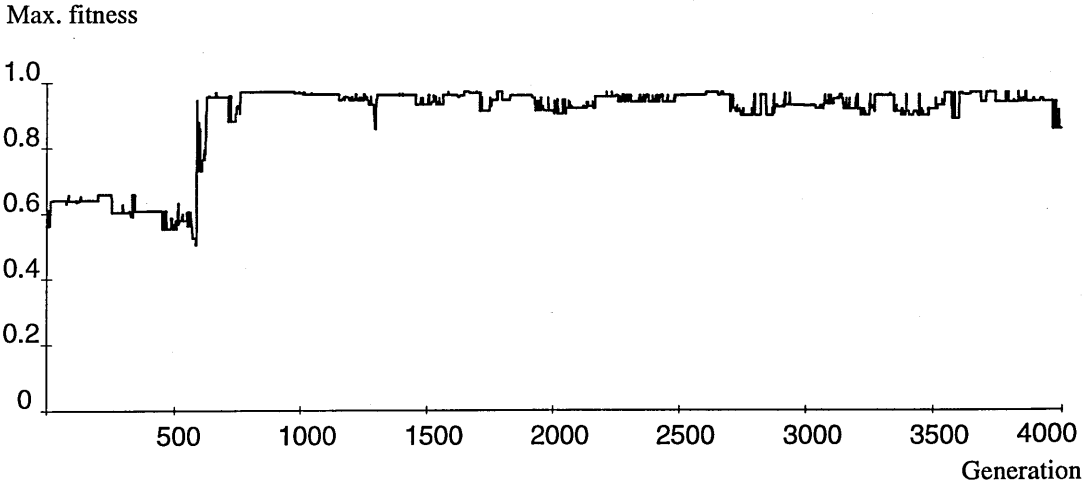


Figure 5.15. Maximum fitness vs. generation number.

Experiment 5.7.4:

Desired Function	f(*v*)=0.45**v*+0.85
Population size	100
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	100

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

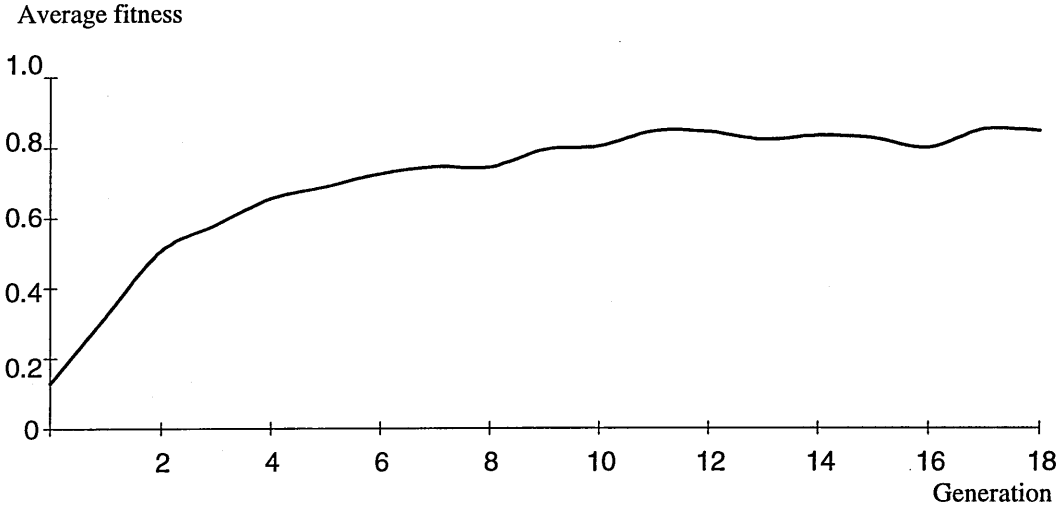


Figure 5.16. Average fitness vs. generation number.

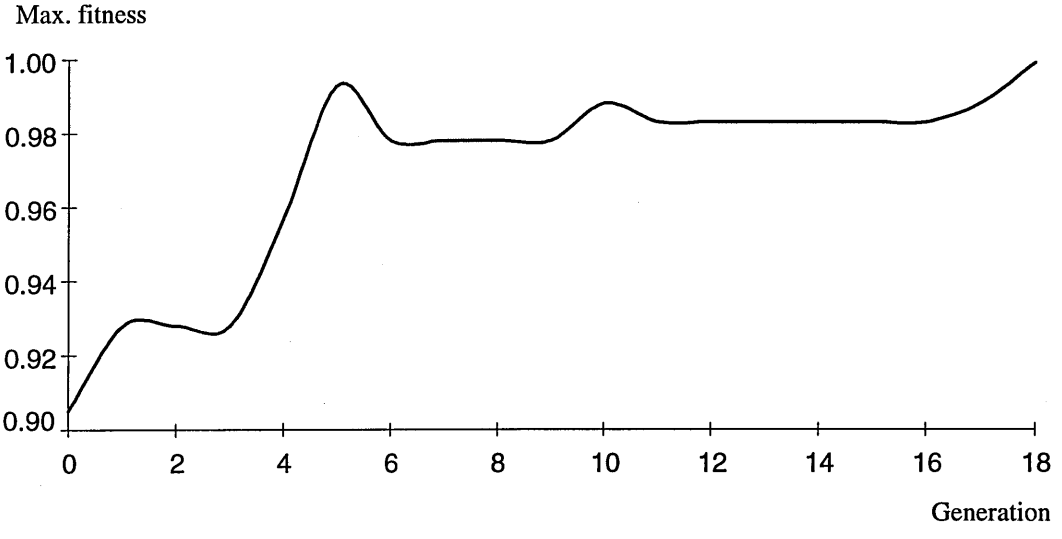


Figure 5.17. Maximum fitness vs. generation number.

Experiment 5.7.5:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	20
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	1000

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

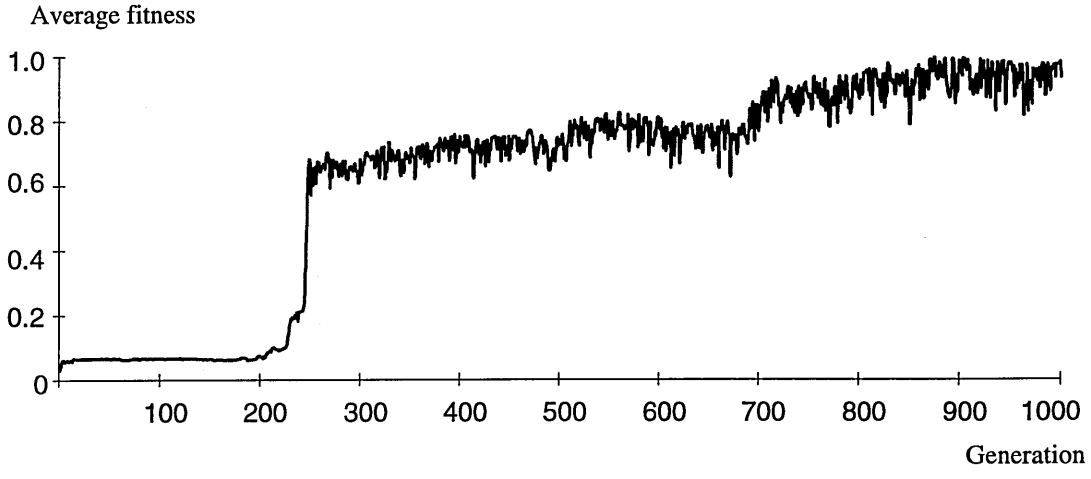


Figure 5.18. Average fitness vs. generation number.

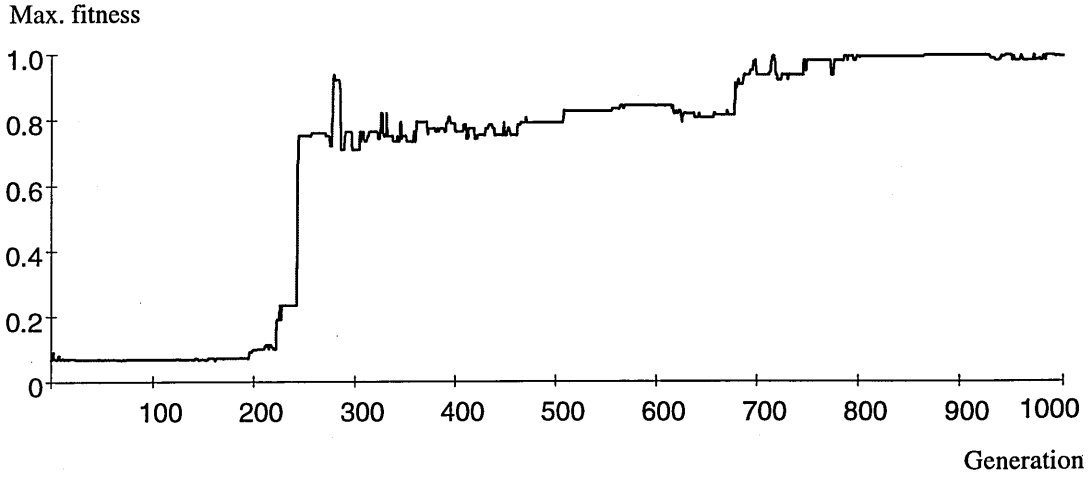


Figure 5.19. Maximum fitness vs. generation number.

Experiment 5.7.6:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	20
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	4000

Fitness function:

$f = |correlation_coef.|$

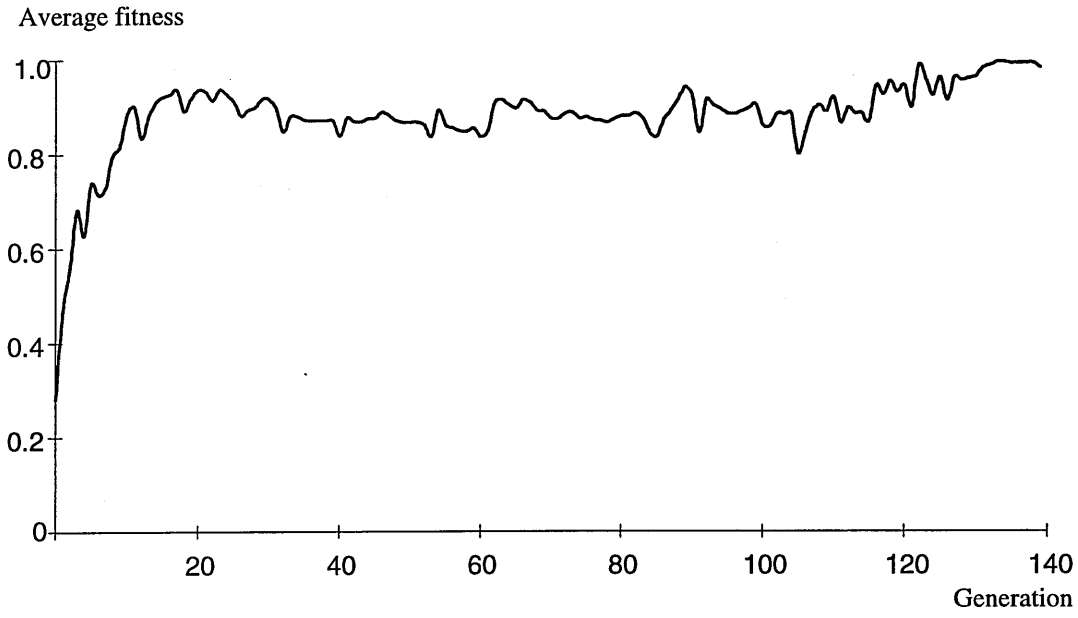


Figure 5.20. Average fitness vs. generation number.

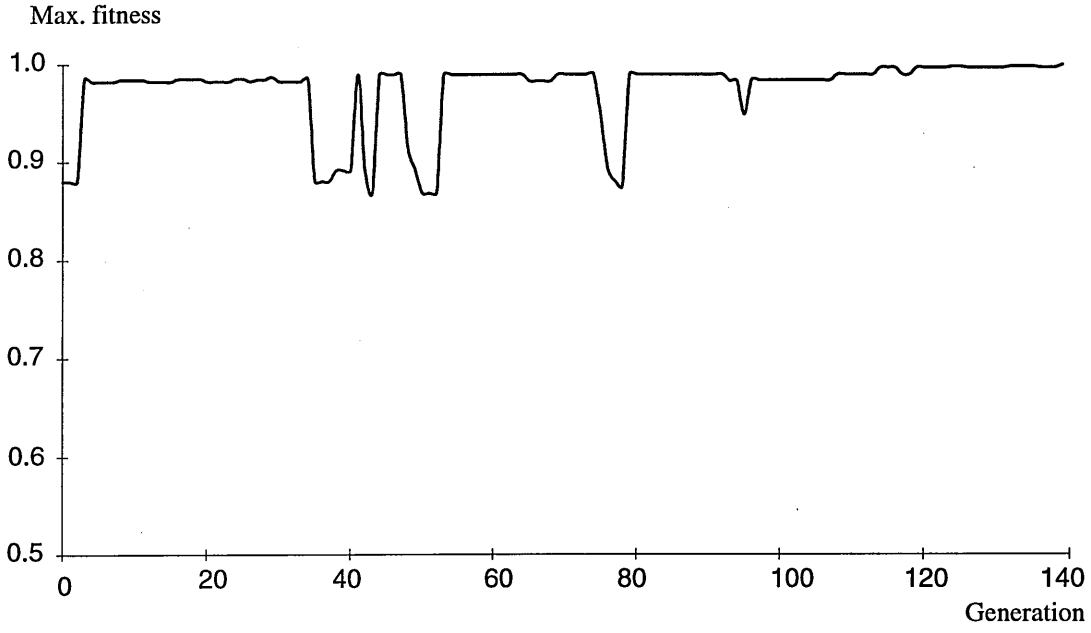


Figure 5.21. Maximum fitness vs. generation number.

Experiment 5.7.7:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	200
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	1000

Fitness function:

$$f = \frac{1}{1 + \sum error_i^2}$$

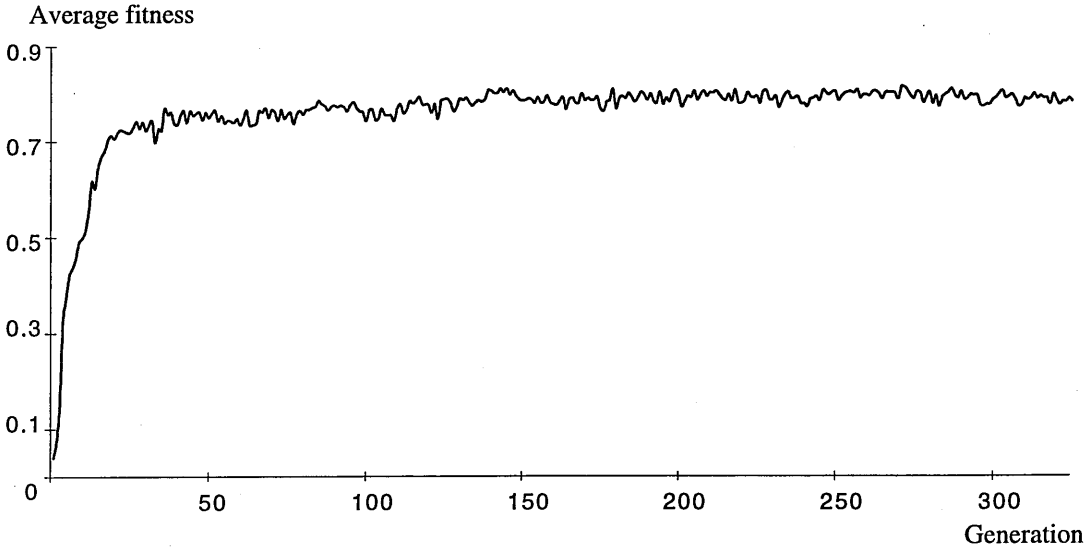


Figure 5.22. Average fitness vs. generation number.

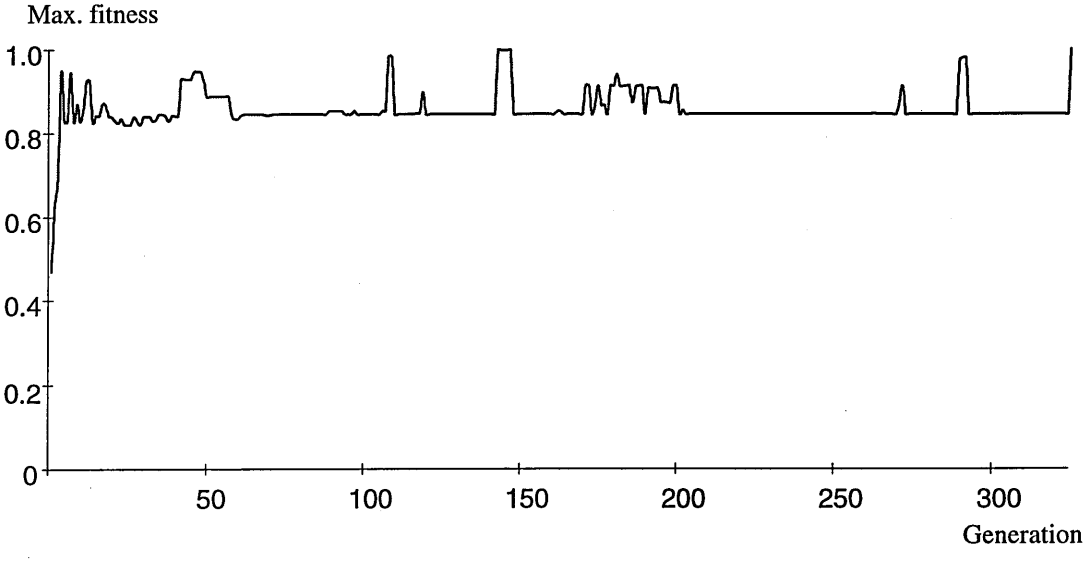


Figure 5.23. Maximum fitness vs. generation number.

Experiment 5.7.8:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	200
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	4000

Fitness function:

$f=|correlation_coef.|$

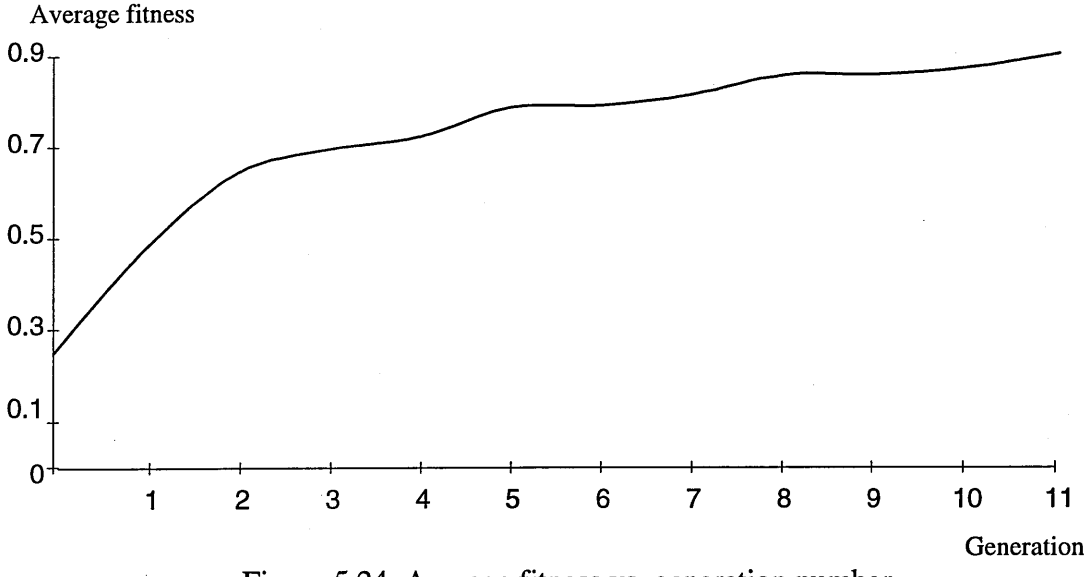


Figure 5.24. Average fitness vs. generation number.

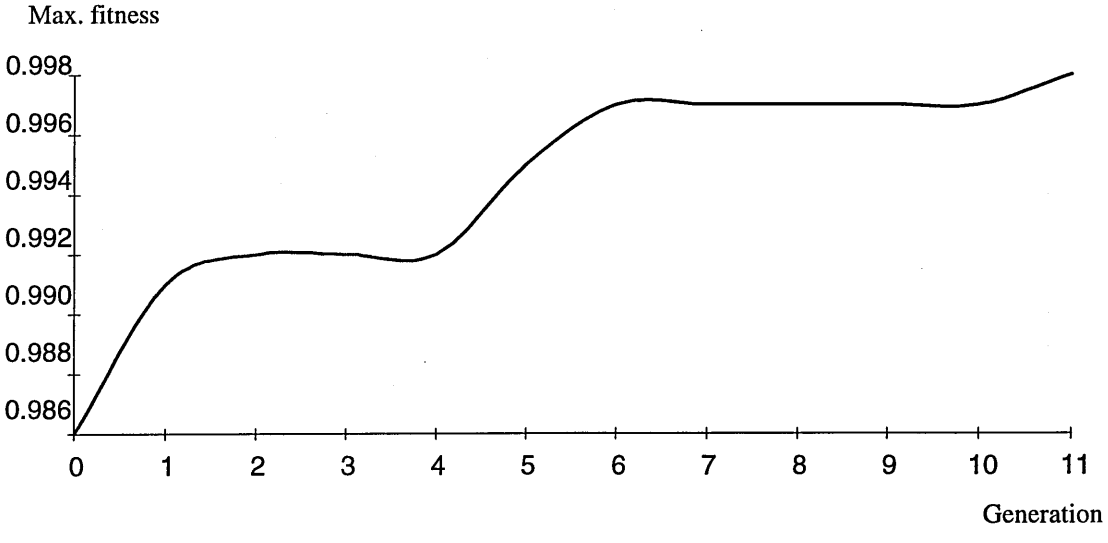


Figure 5.25. Maximum fitness vs. generation number.

Experiment 5.7.9:

Desired function	f(v)= (A+**n**)*Sin(((B+**n**)*v+(C+**n**))*2*π)*Cos((D+**n**)*v)
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	2000

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

Generation	Mean Abs. error	Fitness	A	B	C	D
initial	0.328	0.202	0.055	0.442	0.024	0.05
1000	0.060	0.746	0.682	1.6	0.616	1.308
2000	0.022	0.951	0.686	1.483	0.655	1.382

Table 5.2. The coefficients of the best individuals.

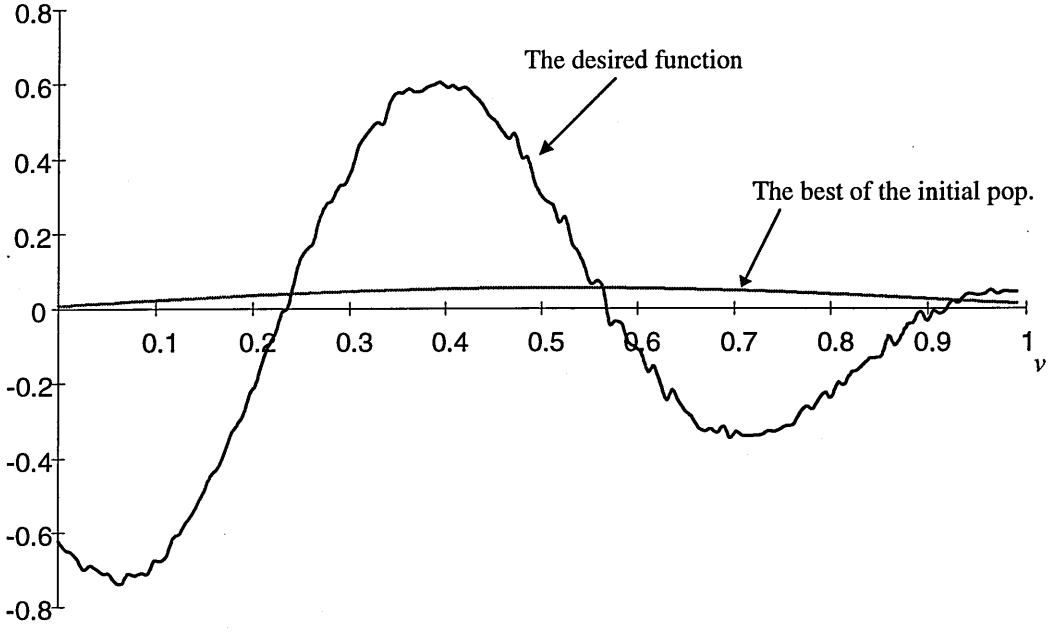


Figure 5.26. The desired function and the best of the initial population

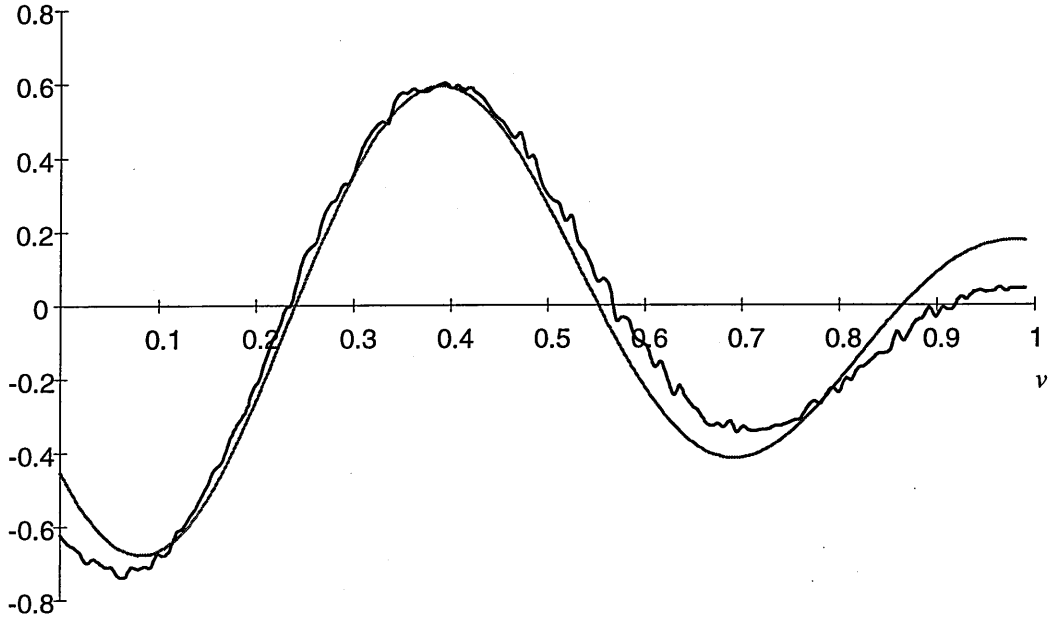


Figure 5.27. The desired function and the best of the 1000th generation.

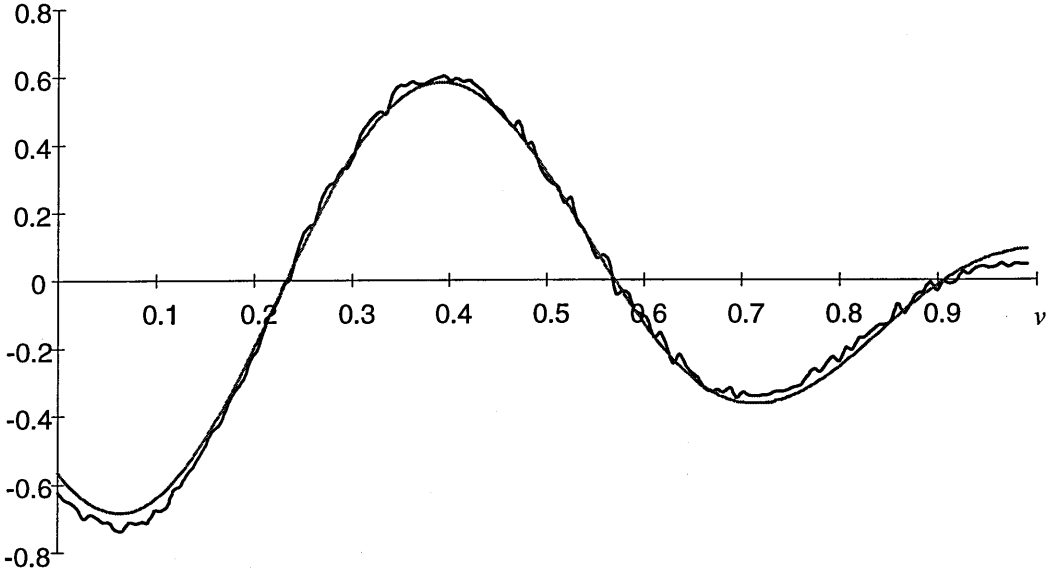


Figure 5.28. The desired function and the best of the 2000th generation.

5.8. TESTING the MODEL GAs-2

Experiment 5.8.1:

Desired function	f(v)=0.243
Population size	2
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	1000

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

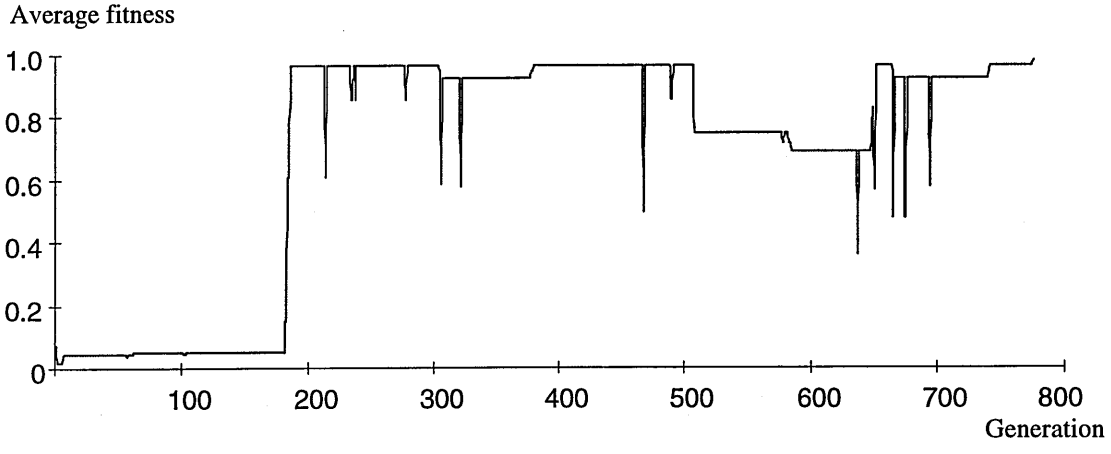


Figure 5.29. Average fitness vs. generation number.

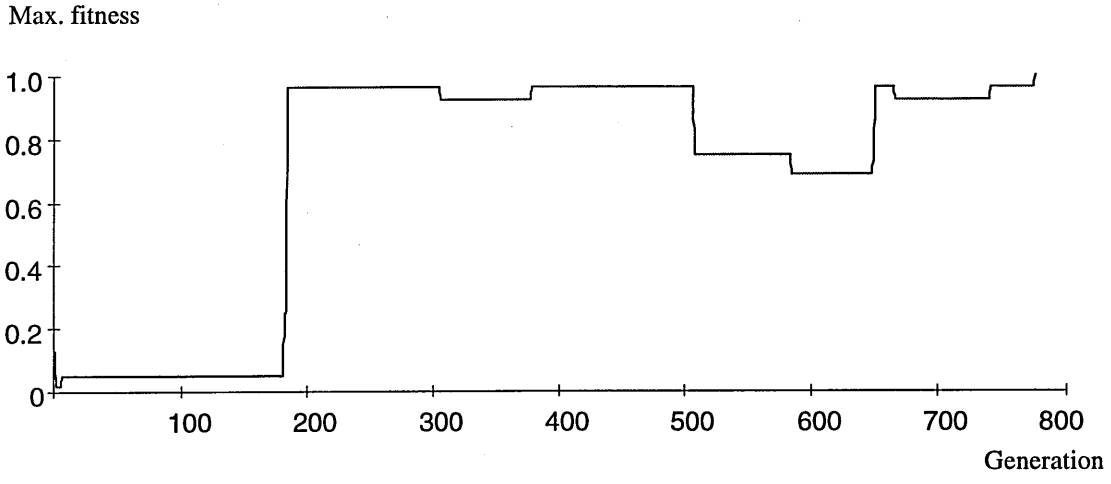


Figure 5.30. Maximum fitness vs. generation number.

Experiment 5.8.2:

Desired function	f(v)=0.243
Population size	10
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	1000

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

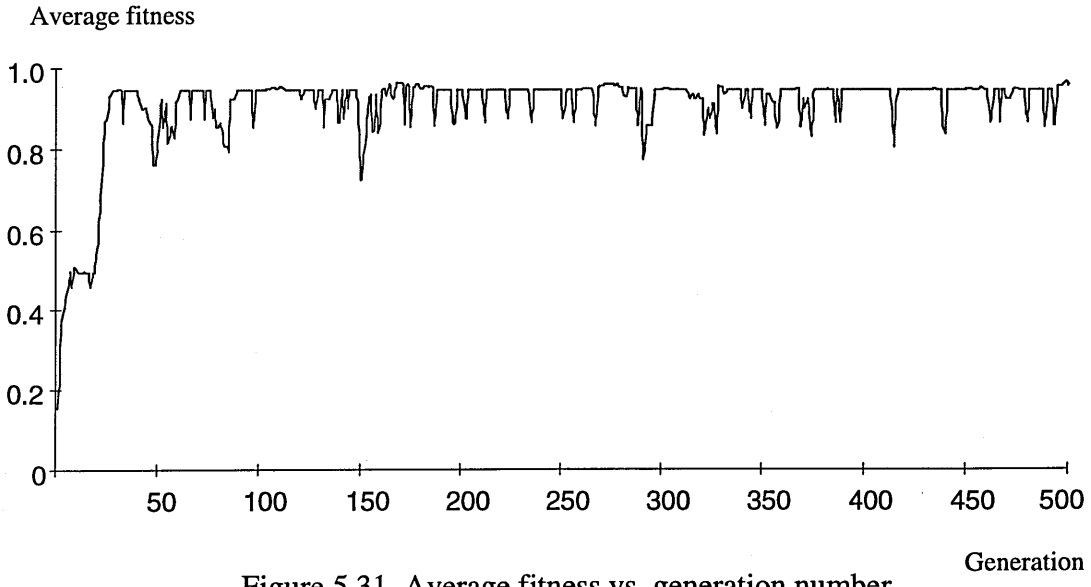


Figure 5.31. Average fitness vs. generation number.

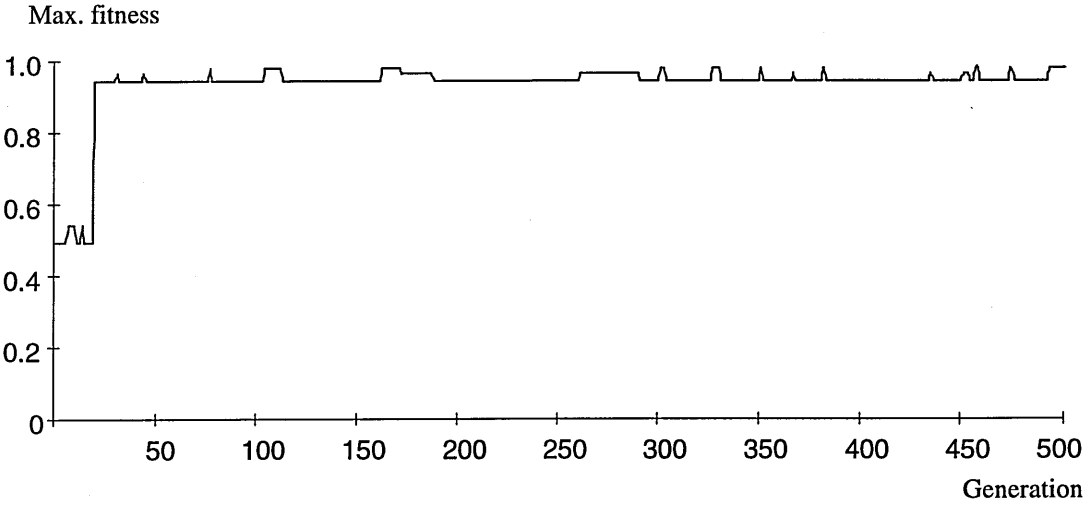


Figure 5.32. Maximum fitness vs. generation number.

Experiment 5.8.3:

Desired Function	f(v)=0.45*v+0.85
Population size	10
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	4000

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

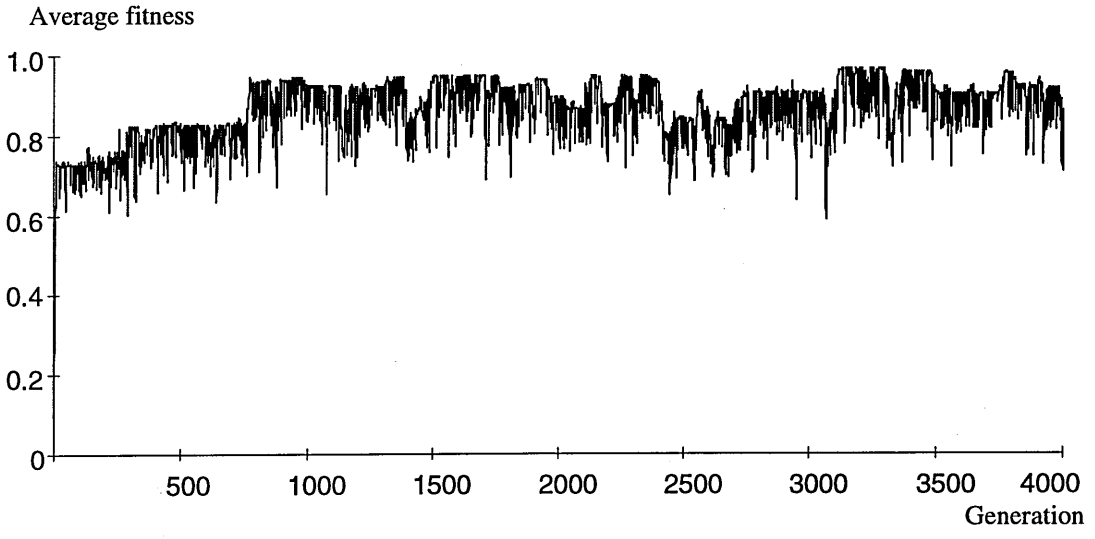


Figure 5.33. Average fitness vs. generation number.

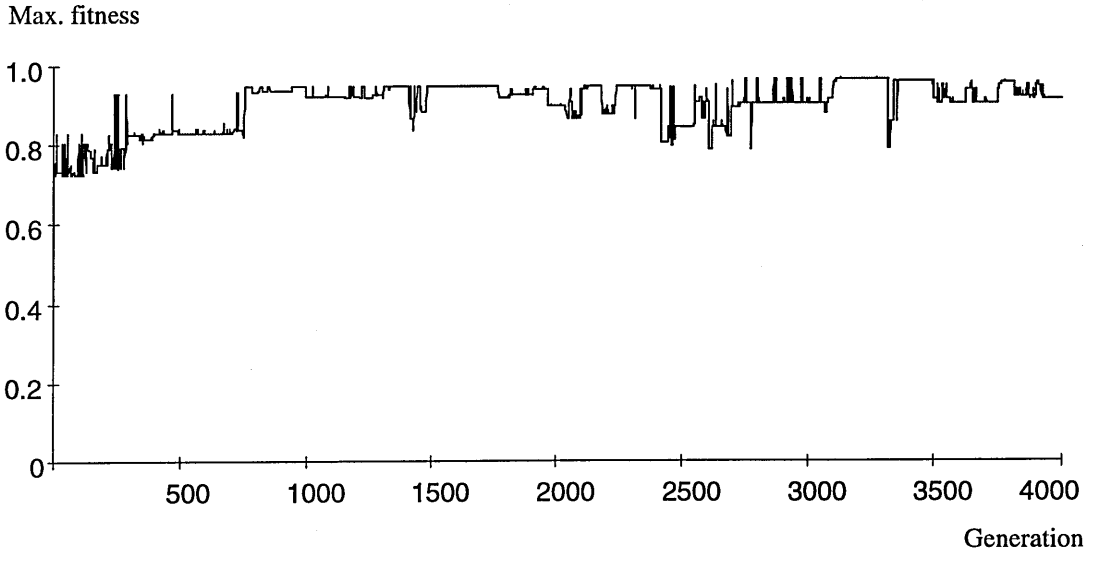


Figure 5.34. Maximum fitness vs. generation number.

Experiment 5.8.4:

Desired Function	f(v)=0.45*v+0.85
Population size	100
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	100

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

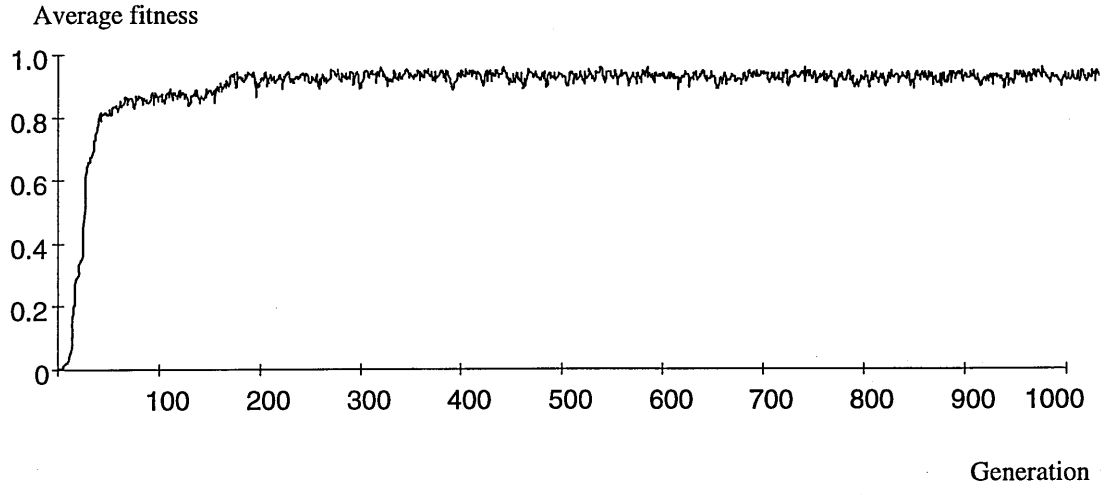


Figure 5.35. Average fitness vs. generation number.

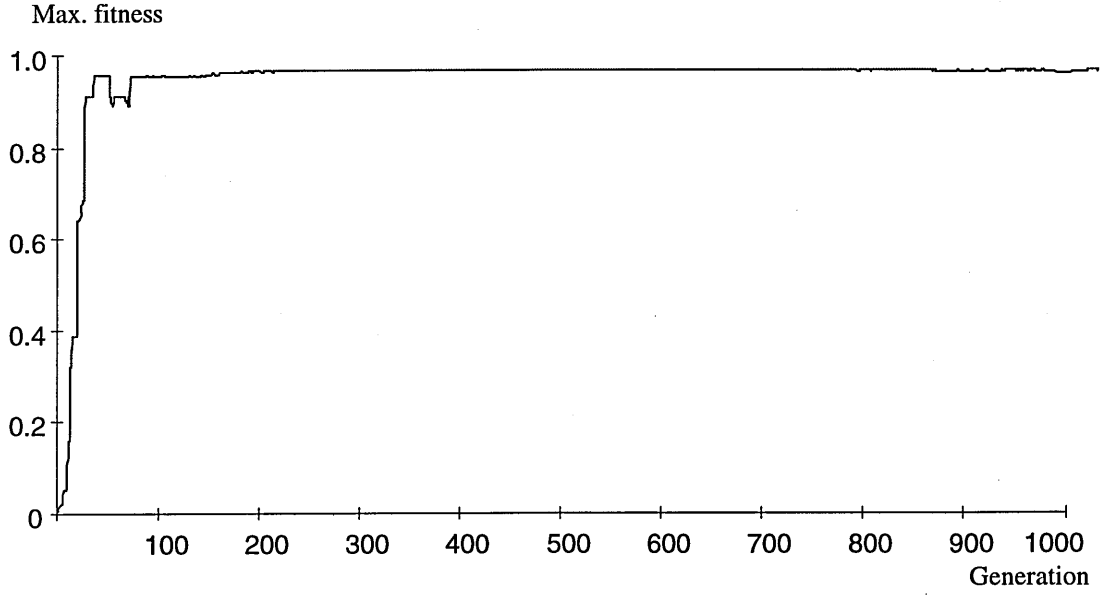


Figure 5.36. Maximum fitness vs. generation number.

Experiment 5.8.5:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	20
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	2000

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

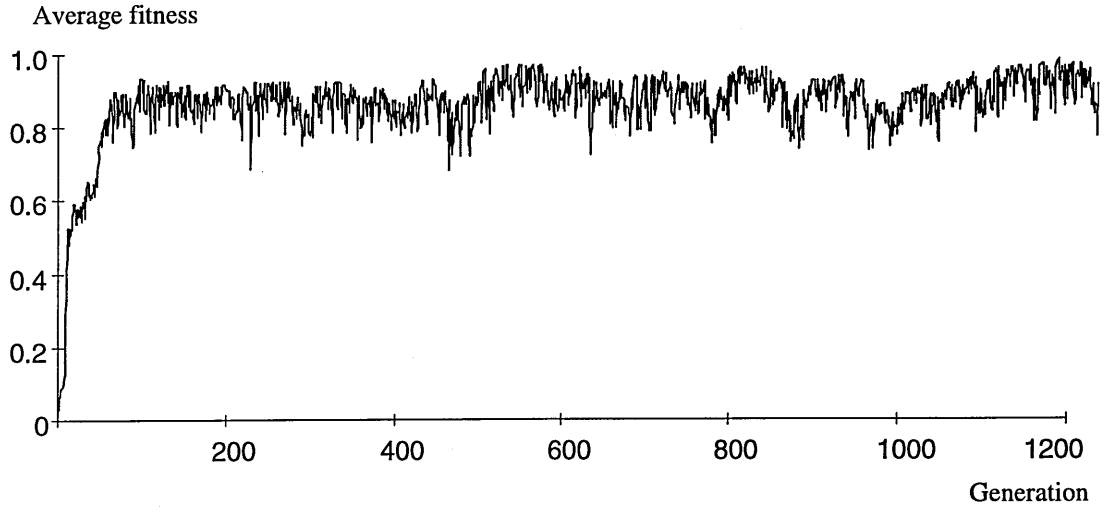


Figure 5.37. Average fitness vs. generation number.

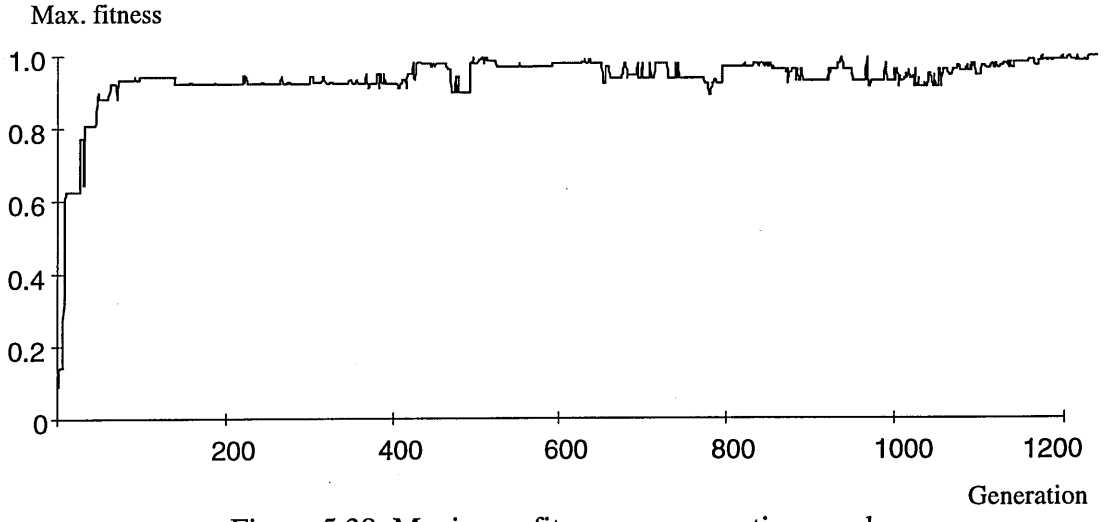


Figure 5.38. Maximum fitness vs. generation number.

Experiment 5.8.6:

Desired function	f(*v*)=0.43*sin(4**v***π+1.88)
Population size	200
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	1000

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

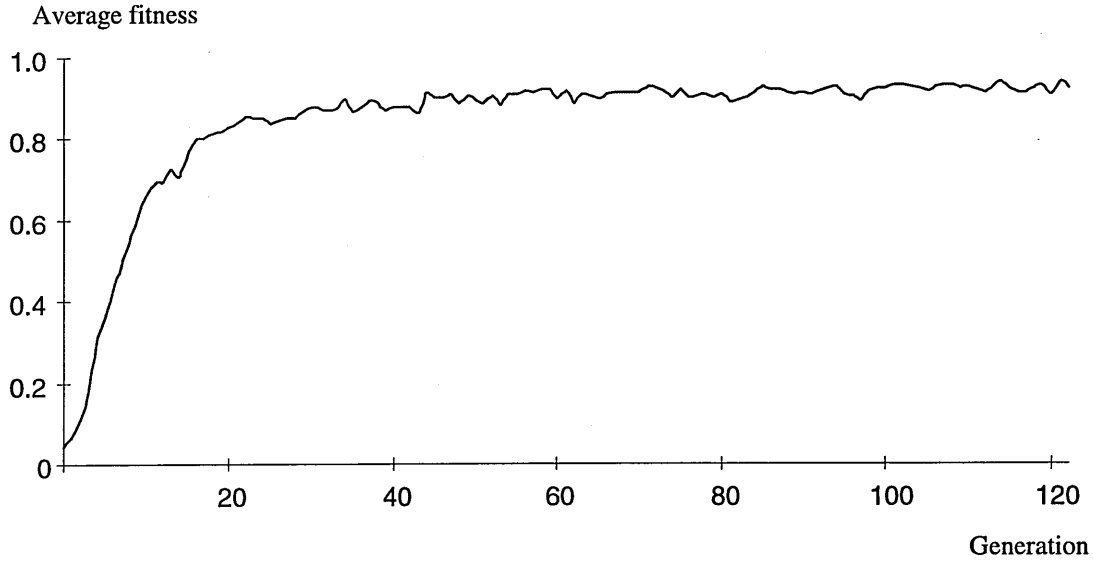


Figure 5.39. Average fitness vs. generation number.

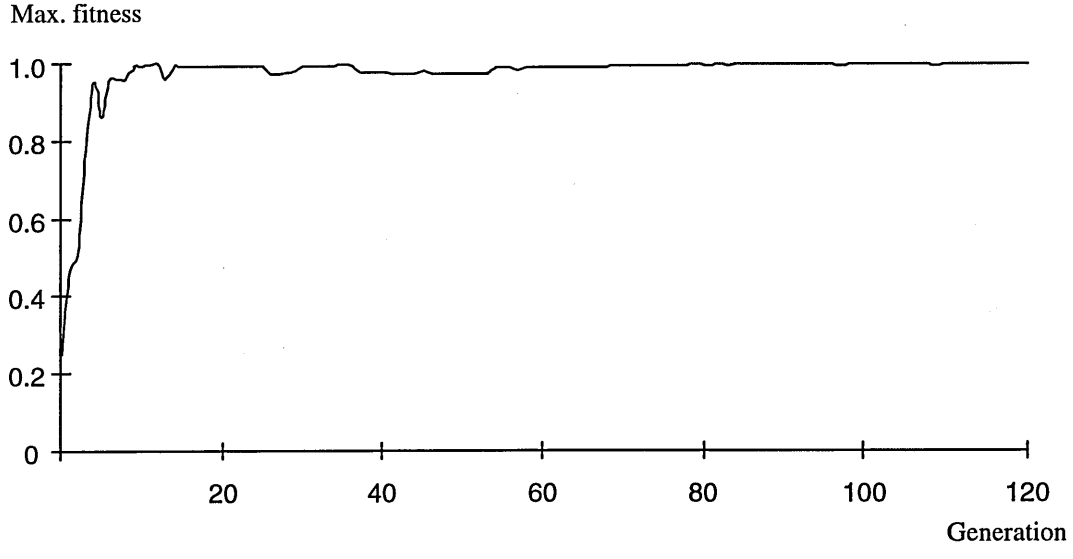


Figure 5.40. Maximum fitness vs. generation number.

Experiment 5.8.7:

Desired function	f(*v*)= (A+**n**)*Sin(((B+**n**)*v+(C+**n**))*2*π)*Cos((D+**n**)*v)
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	2000

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

Generation	Mean Abs. error	Fitness	A	B	C	D
initial	0.327	0.202	0.051	0.450	0.027	0.058
1000	0.031	0.915	0.753	1.483	0.655	1.392
2000	0.031	0.915	0.753	1.483	0.655	1.392

Table 5.3. The coefficients of the best individuals.

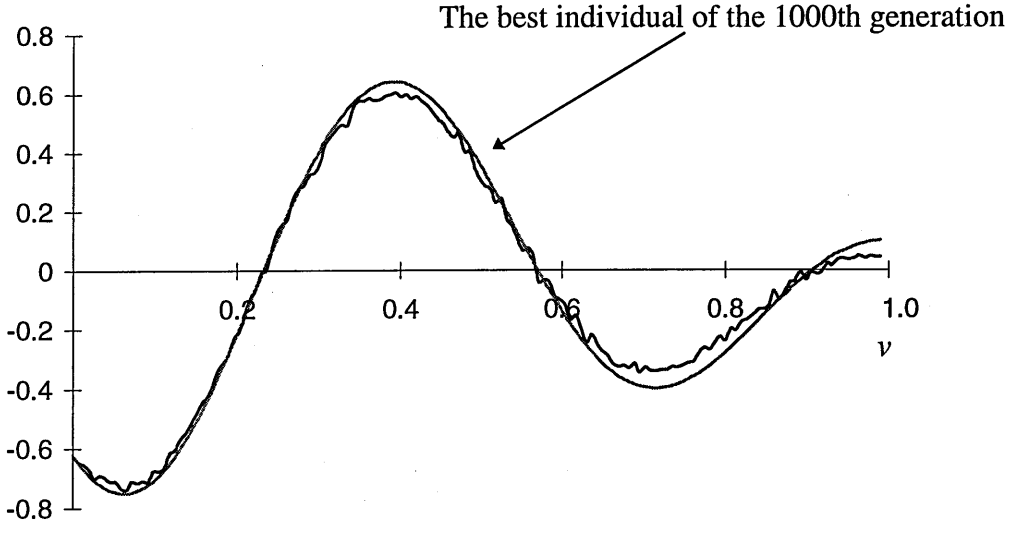


Figure 5.41. The desired function and the best of the 1000th generation.

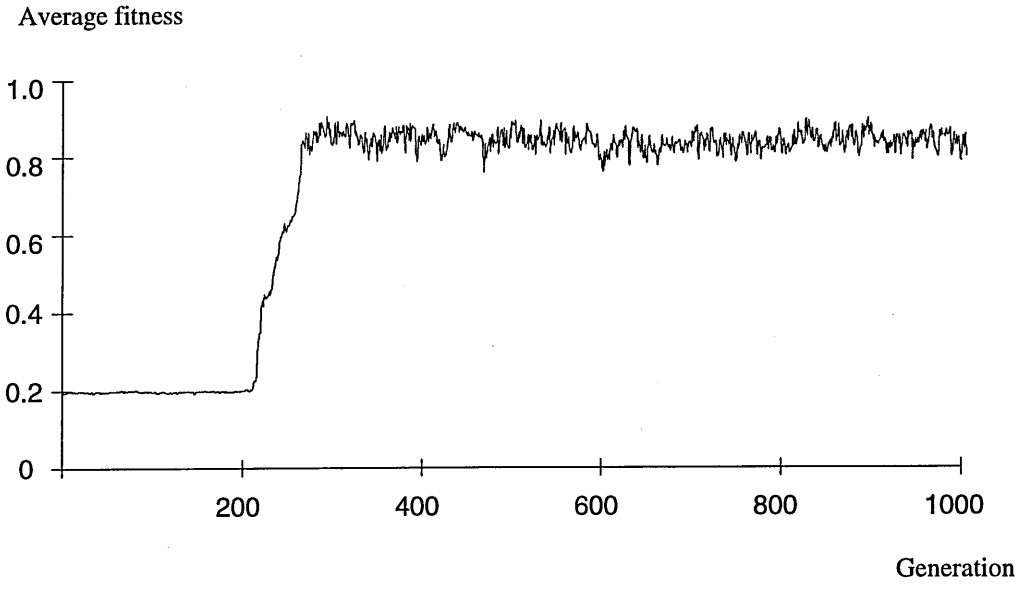


Figure 5.42. Average fitness vs. generation number.

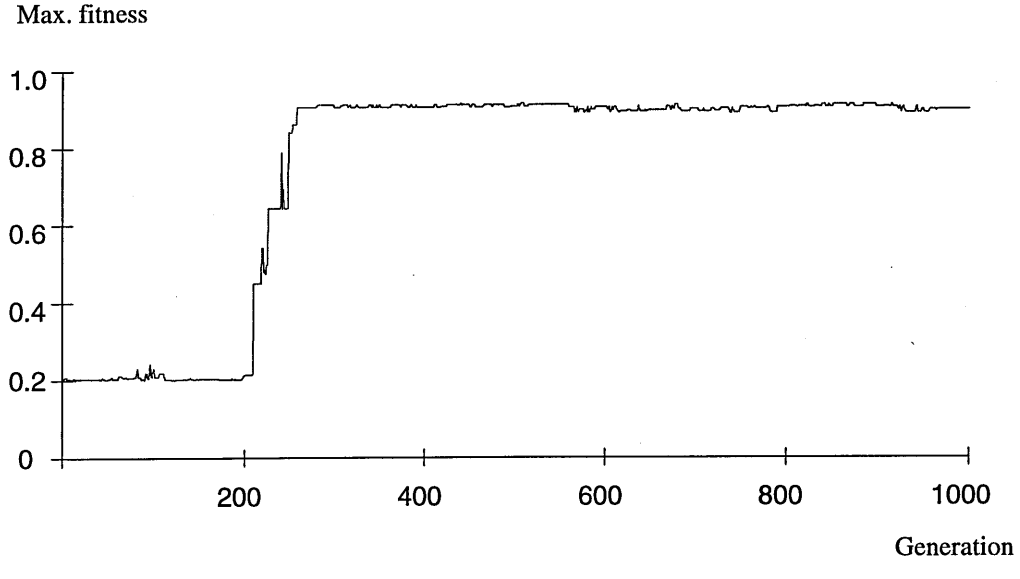


Figure 5.43. Maximum fitness vs. generation number.

Experiment 5.8.8:

Desired function	f(v)= (A+**n**)*Sin(((B+**n**)*v+(C+**n**))*2*π)*Cos((D+**n**)*v)
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Max. Generation	2000

Fitness function:

0-86th gen.:

$$f=|correlation_coef.|$$

87th-437th gen.

$$f=\frac{|correlation_coef.|}{1+\sum error_i^2}$$

Generation	Mean Abs. Error	Fitness	A	B	C	D
initial	0.356	0.627	0.114	0.133	0.165	0.350
86	0.235	0.984	0.220	1.308	0.729	1.550
2000	0.020	0.957	0.702	1.458	0.667	1.417

Table 5.4. The coefficients of the best individuals.

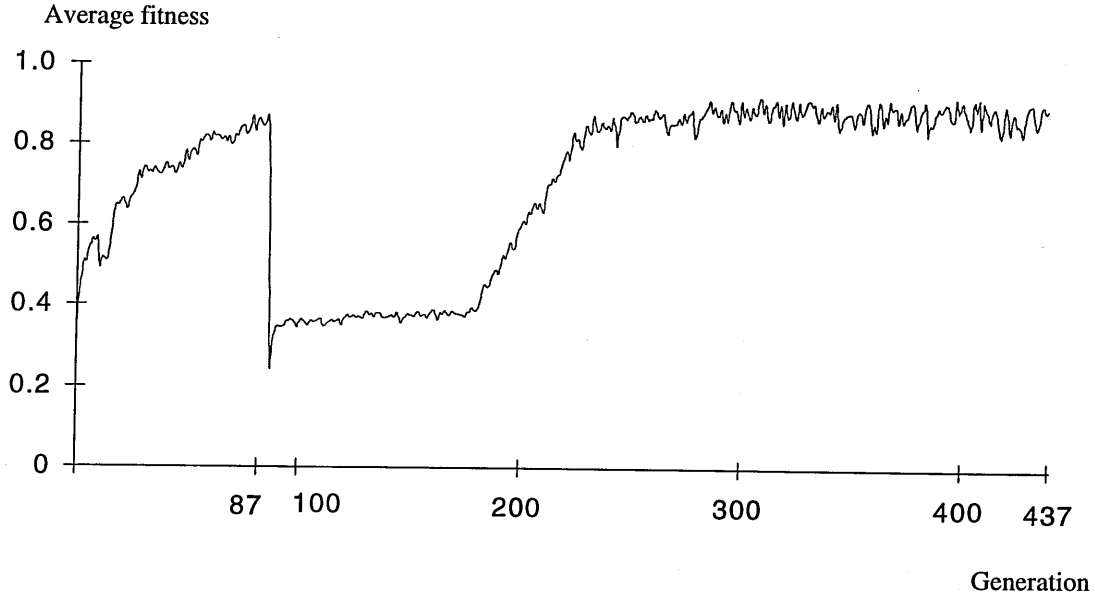


Figure 5.44. Average fitness vs. generation number.

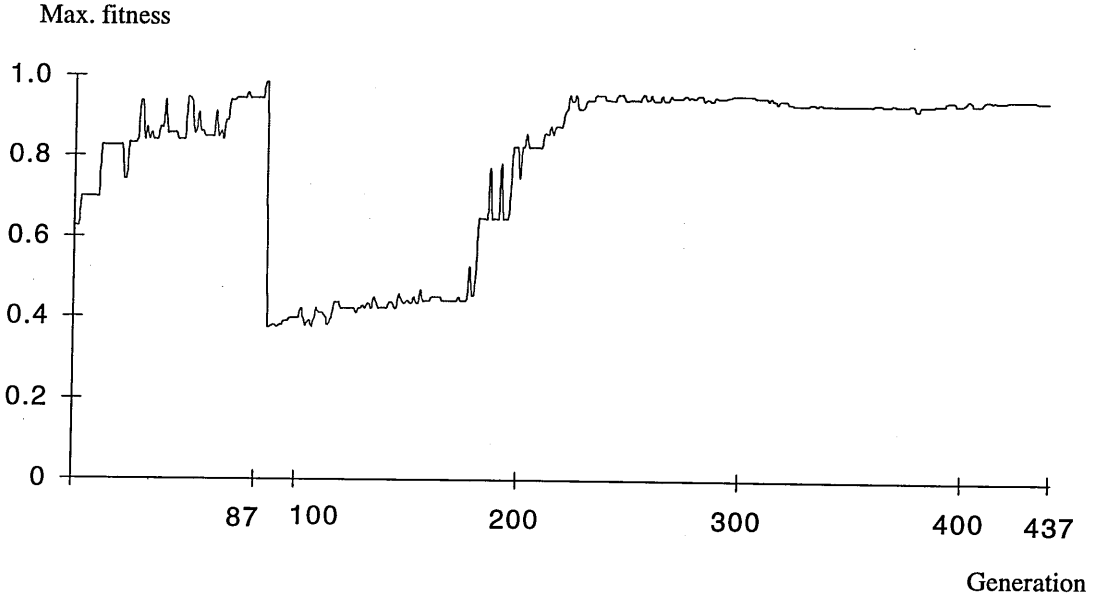


Figure 5.45. Maximum fitness vs. generation number.

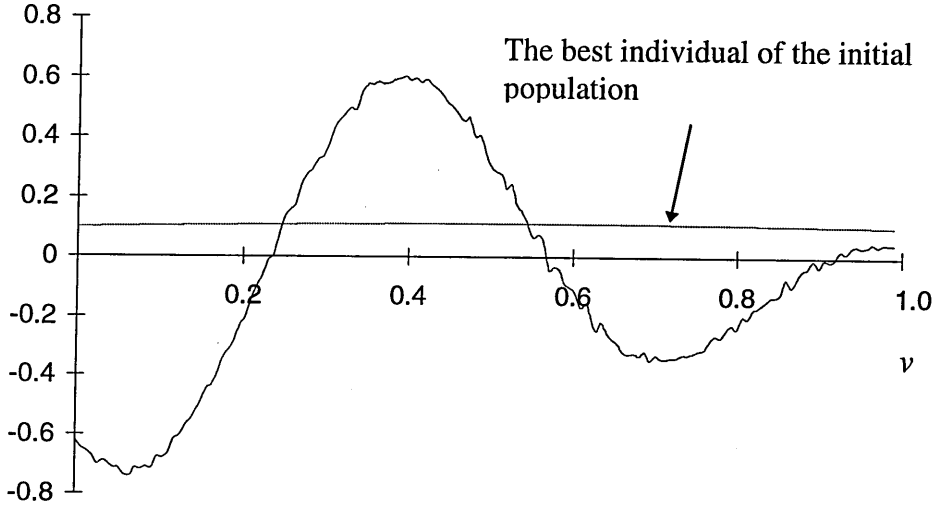


Figure 5.46. The desired function and the best of the initial population

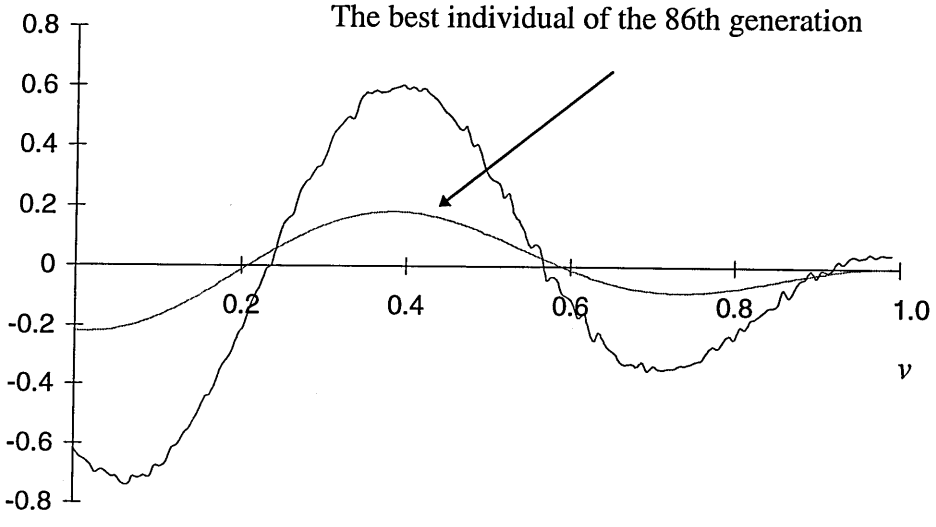


Figure 5.47. The desired function and the best of the 86th generation.

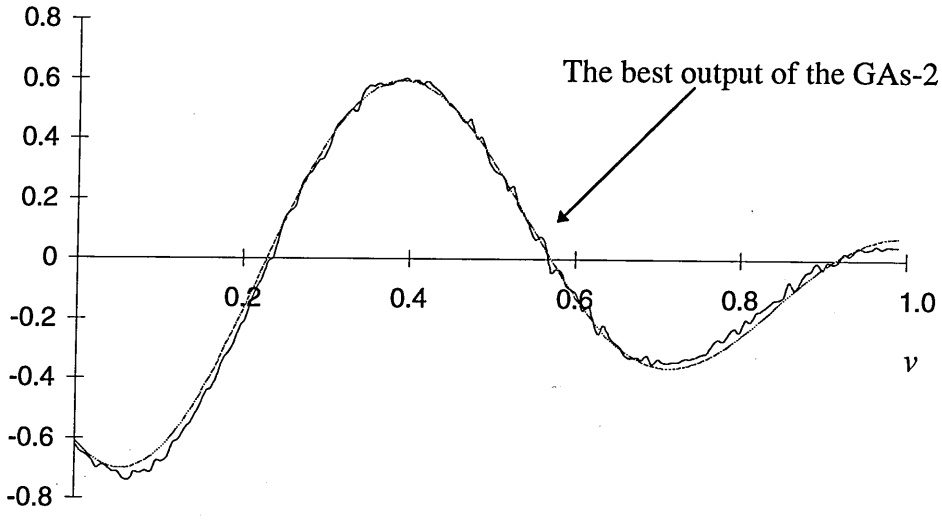


Figure 5.48. The desired function and the best output of the GAs-2.

5.9. TESTING The MODEL GP-1

- The Function *Constant*

Experiment 5.9.1:

Desired function	f(v)=0.243
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	48
Max. Generation	100

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

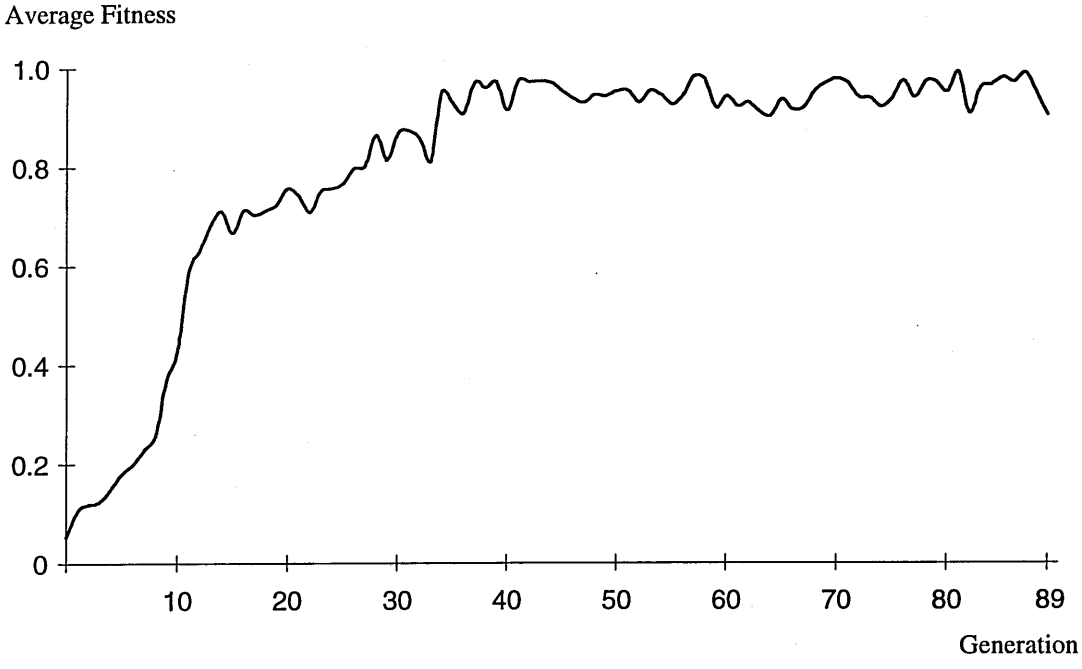


Figure 5.49. The average fitness against generation number.

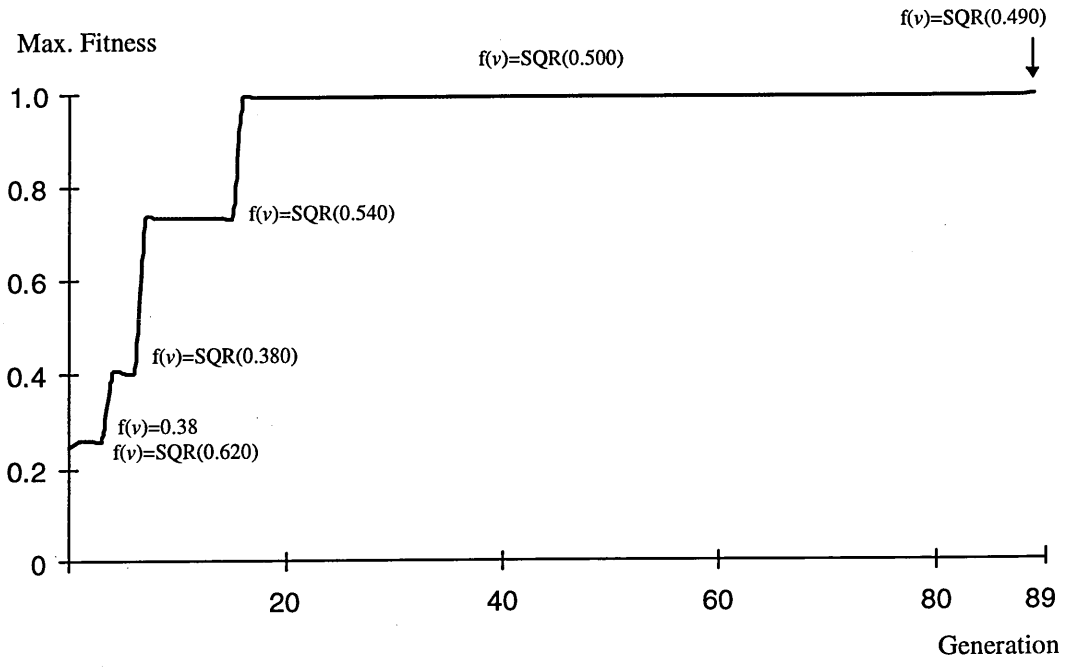


Figure 5.50. The maximum fitness against generation number.

Experiment 5.9.2:

Desired function	f(v)=0.243
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	96
Max. Generation	100

Fitness function:

$$f = \frac{1}{1 + \sum error_i^2}$$

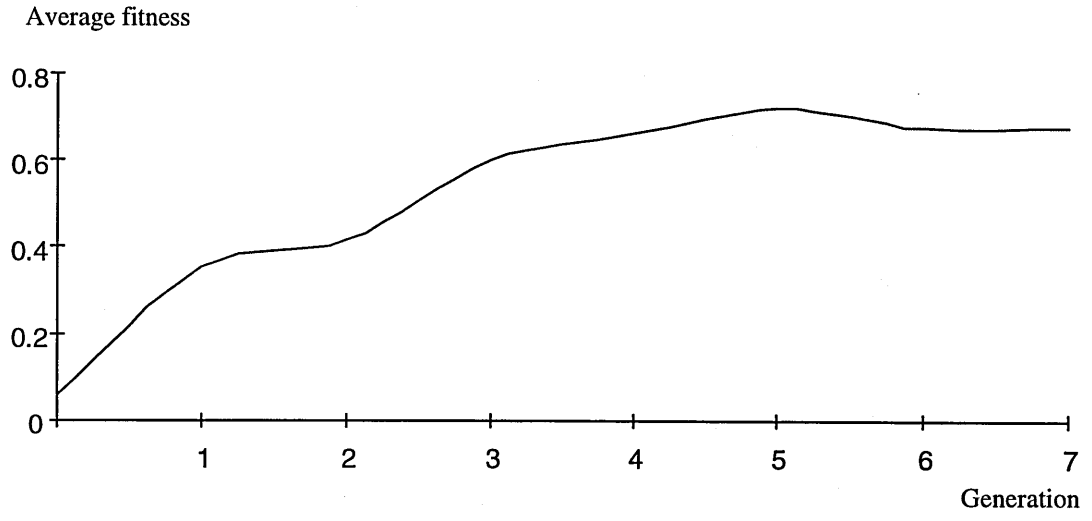


Figure 5.51. The average fitness against generation number

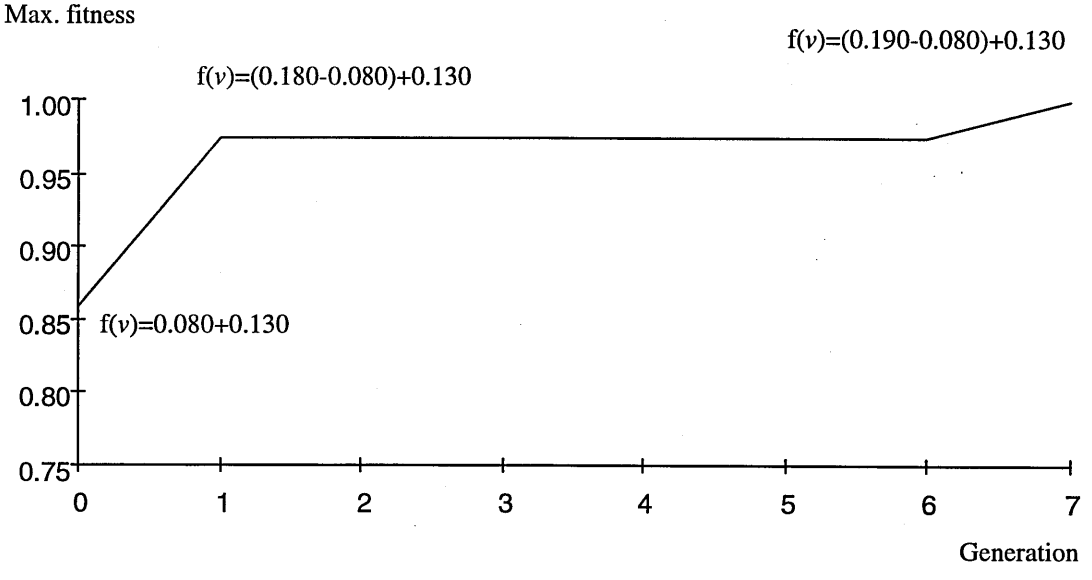


Figure 5.52. The maximum fitness against generation number.

Experiment 5.9.3:

Desired function	f(v)=0.243
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	192
Max. Generation	100

Fitness function:

$$f = \frac{1}{1 + \sum error_i^2}$$

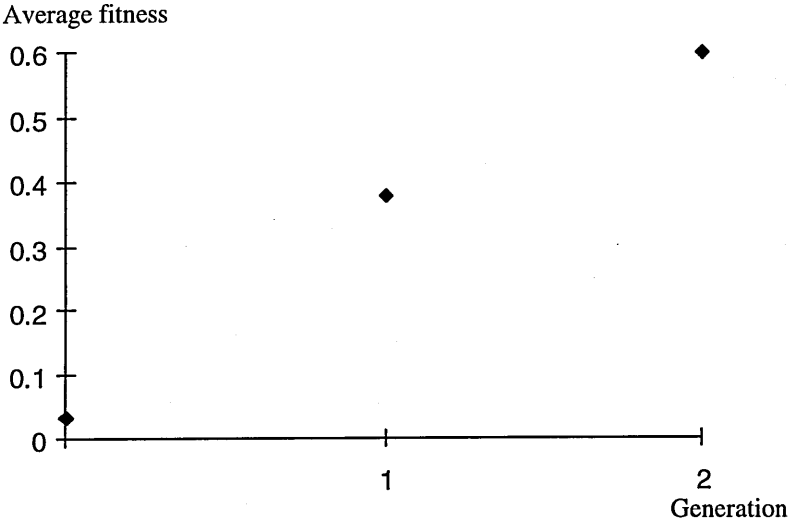


Figure 5.53. The average fitness against generation number.

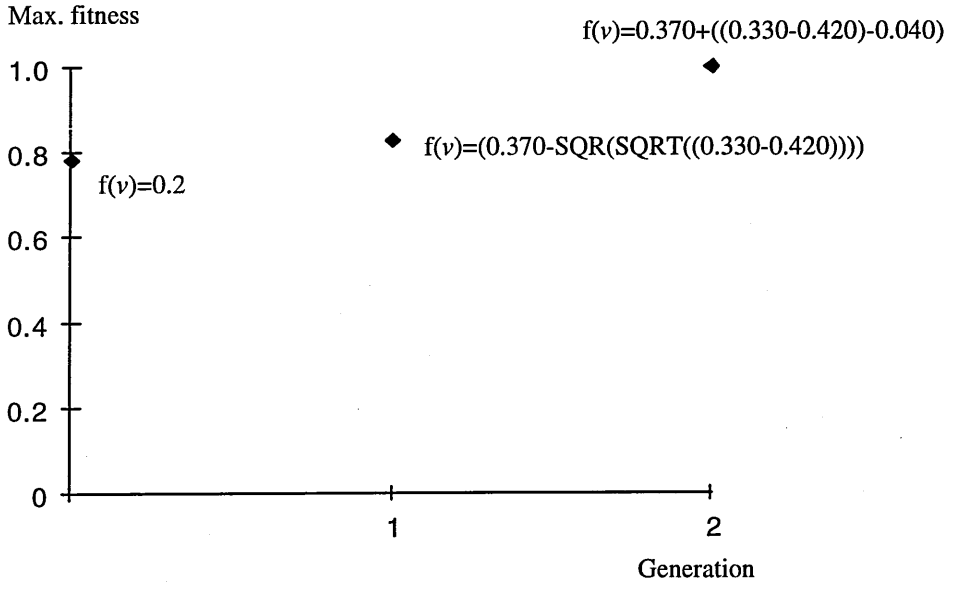


Figure 5.54. The maximum fitness against generation number.

• The Function *Linear*

Experiment 5.9.4:

Desired function	f(*v*)=0.45**v*+0.85
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	96
Max. Generation	100

Fitness function:

$$f = \frac{1}{1 + \sum error_i^2}$$

The best of 100 generations was f(*v*)=0.800+0.290.

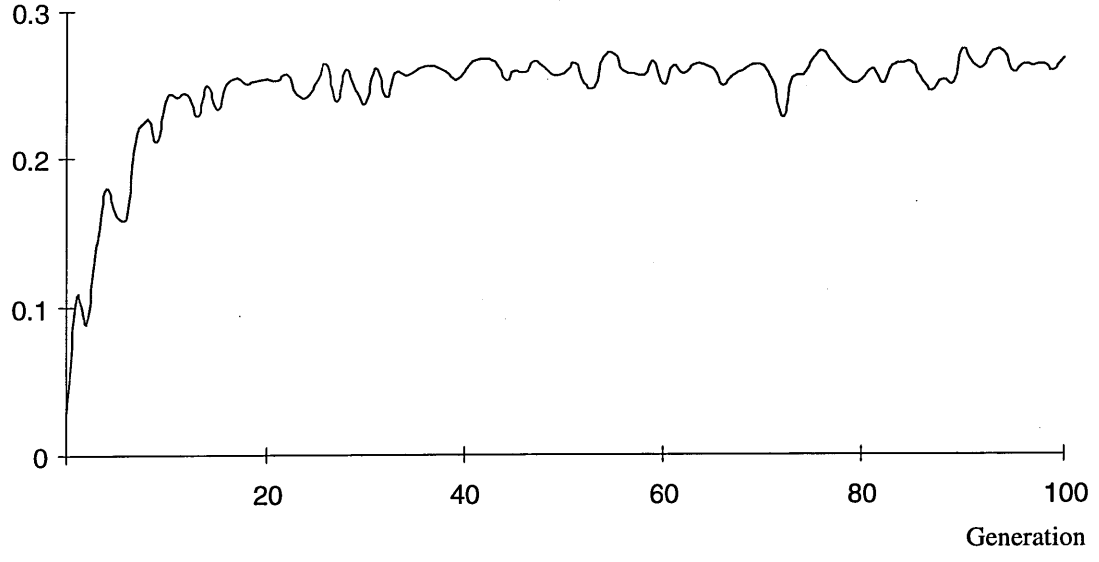


Figure 5.55. The average fitness against generation number.

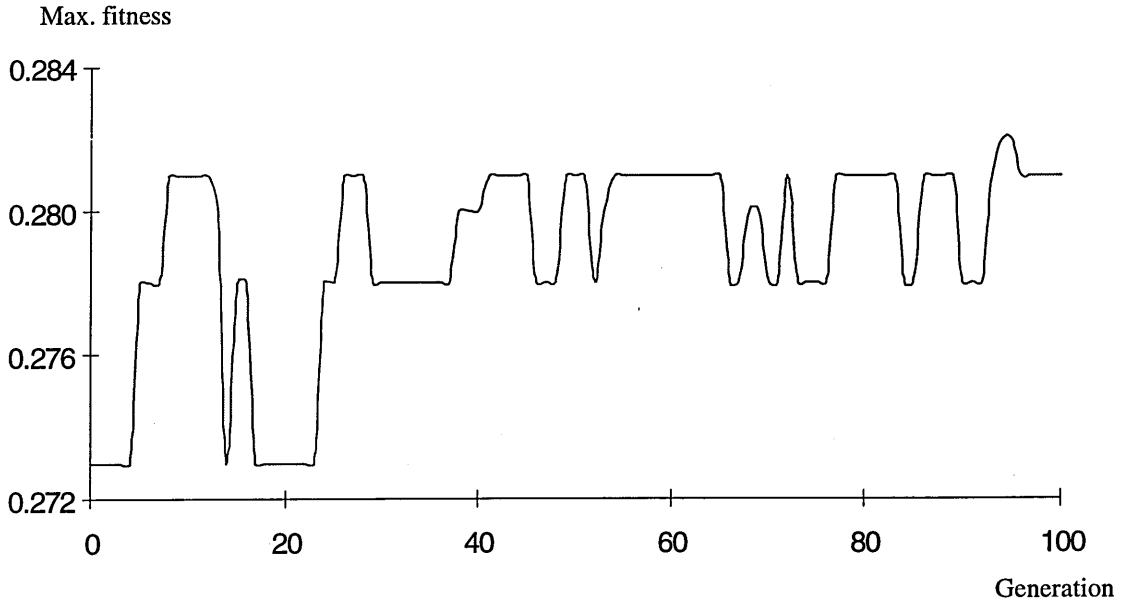


Figure 5.56. The maximum fitness against generation number.

Experiment 5.9.5:

Desired function	f(v)=0.45*v+0.85
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	192
Max. Generation	100

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

The best of 100 generations was f(v)=SQRT(SQRT(SQRT((SQR(v)+1.340)))).

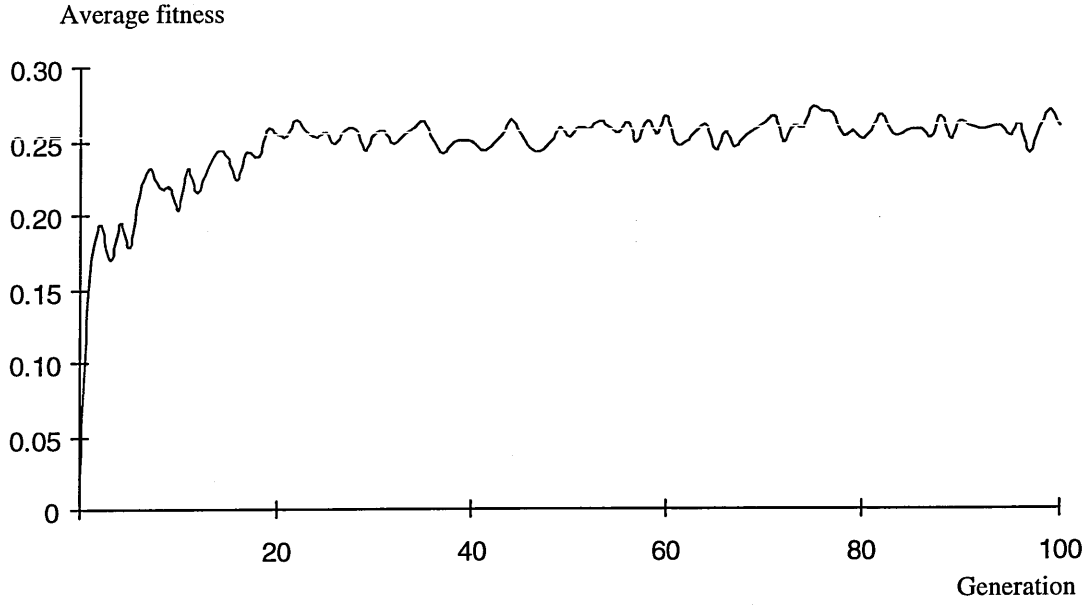


Figure 5.57. The average fitness against generation number.

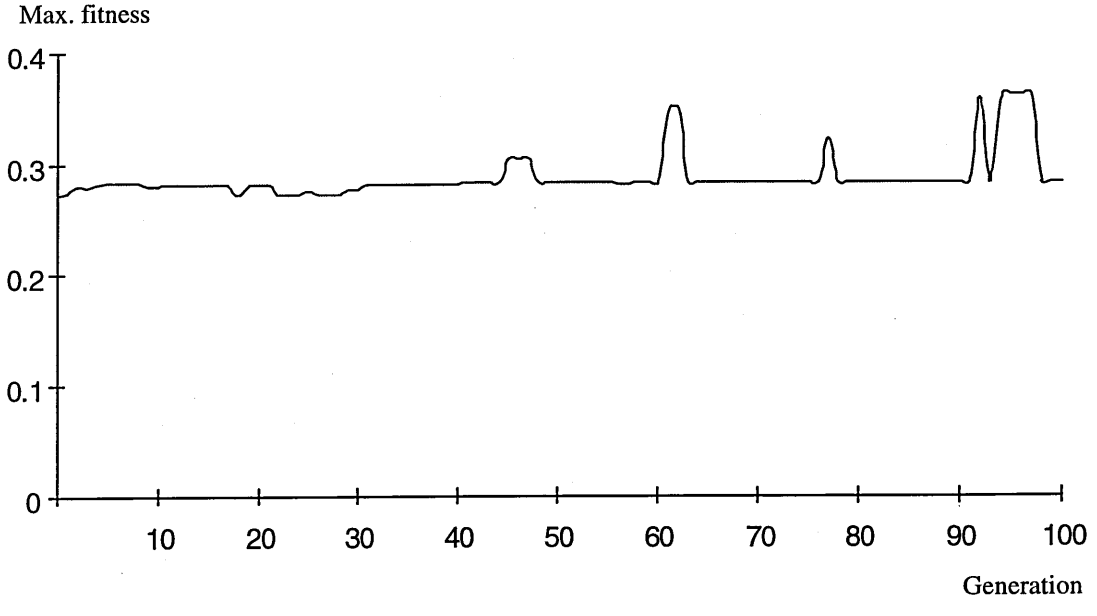


Figure 5.58. The maximum fitness against generation number.

Experiment 5.9.6:

Desired function	f(v)=0.45*v+0.85
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	320
Max. Generation	100

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

The best of 100 generations was

f(v)=0.460+((1.580*SQR(1.970)-SQR(0.490))/SQR(2.030*1.530)).

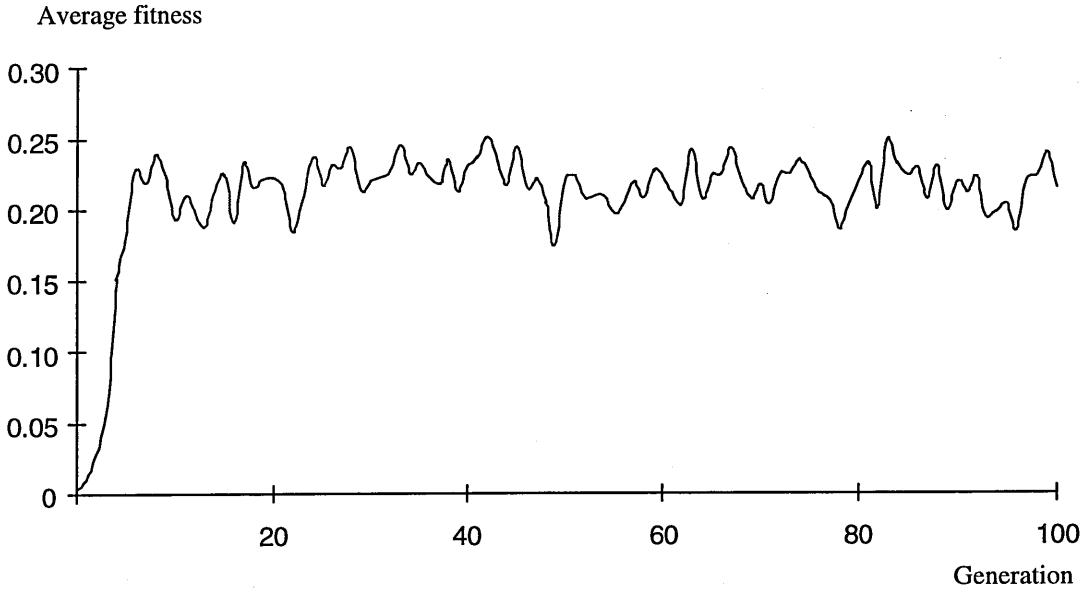


Figure 5.59. The average fitness against generation number.

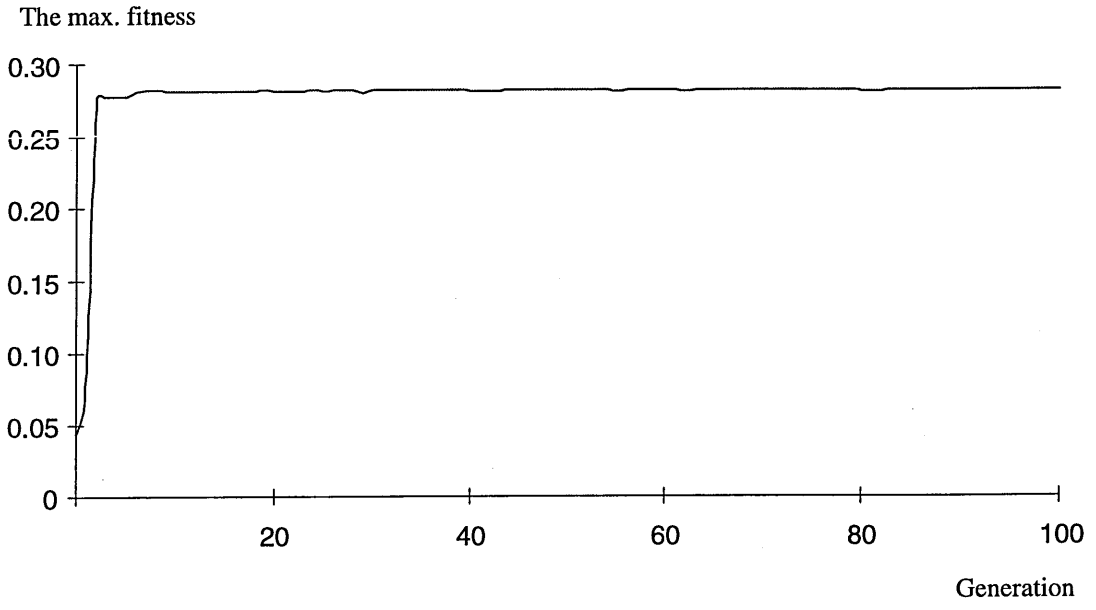


Figure 5.60. The maximum fitness against generation number.

Experiment 5.9.6&7&8:

For the next three experiments, the model GP-1 found the best individual in the initial population.

Desired function	f(v)=0.45*v+0.85
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	96,192,320
Max. Generation	100

Fitness function:

$f = |correlation_coef.|$

Experiment no.	Max. Fitness	The Best Individual	The Length of an Individual
1	1.0	f(v)=(2.460+v)	96
2	1.0	f(v)=1.180+0.470*v+1.560+1.690	192
3	1.0	f(v)=((SQRT(0.430)+0.960)-(v/SQRT(2.210)))+v+1.360+1.690	320

Table 5.5. The results of the three experiments which were undertaken using the correlation coefficient in the fitness calculation.

Experiment 5.9.9:

Desired function	f(*v*)=0.45**v*+0.85
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	96
Max. Generation	100

Fitness function:

$$f = \frac{|correlation_coef.|}{1 + \sum error_i^2}$$

The best individual of 100 generations was f(*v*)=SQRT((*v*+0.680)).

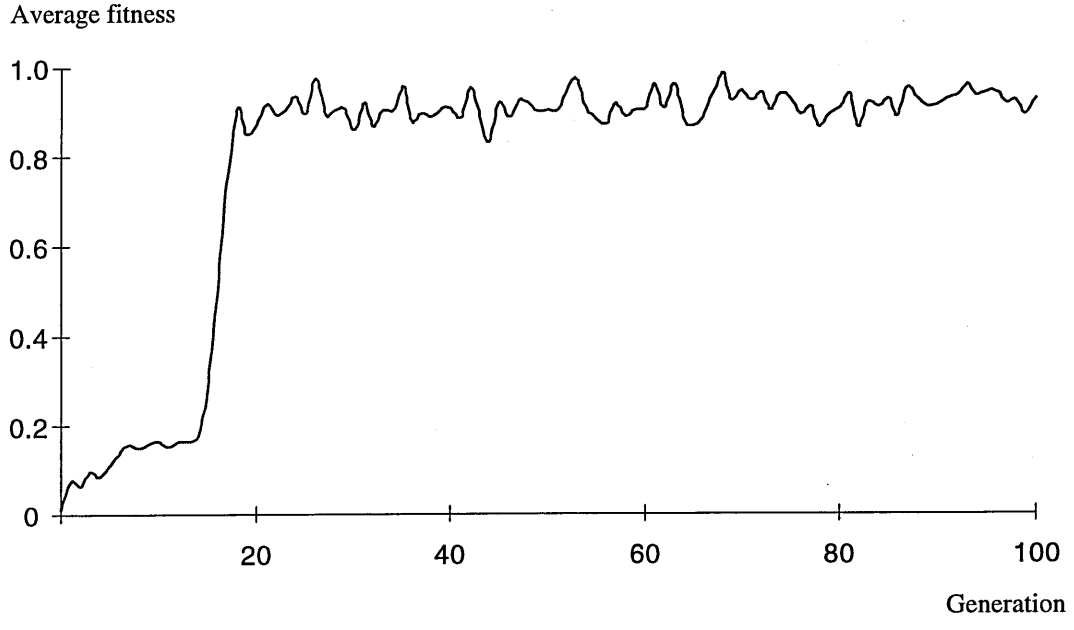


Figure 5.61. The average fitness against generation number.

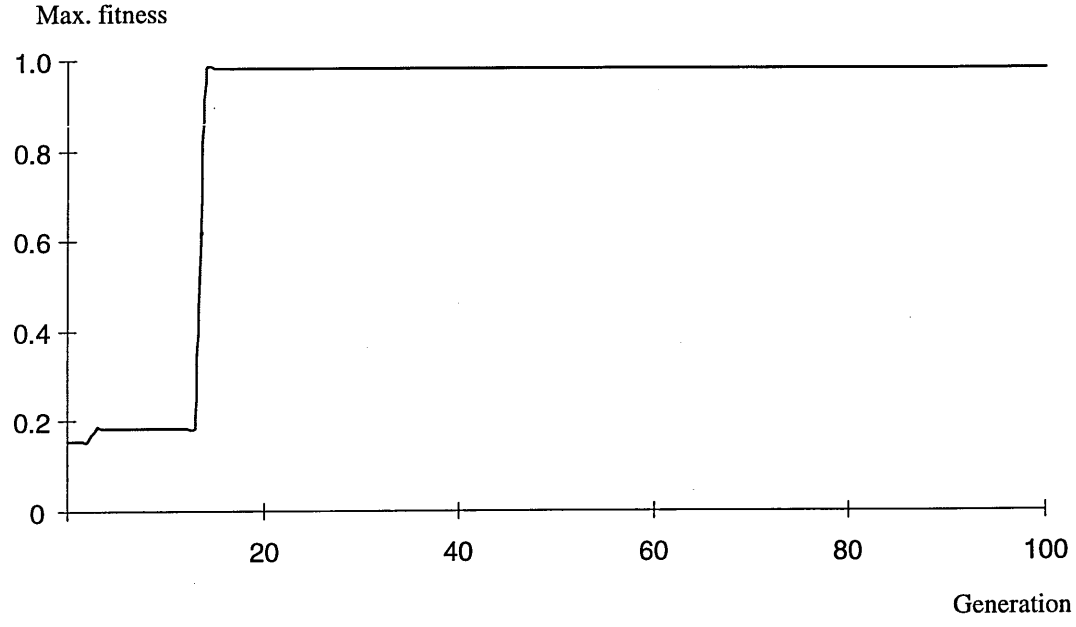


Figure 5.62. The maximum fitness against generation number.

Experiment 5.9.10:

Desired function	f(*v*)=0.45**v*+0.85
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	192
Max. Generation	100

Fitness function:

$$f = \frac{|correlation_coef.|}{1 + \sum error_i^2}$$

The best individual of the 100 generations was

f(*v*)=SQRT(SQR(0.300*SQR(*v*)*SQRT(*v*)))+1.000

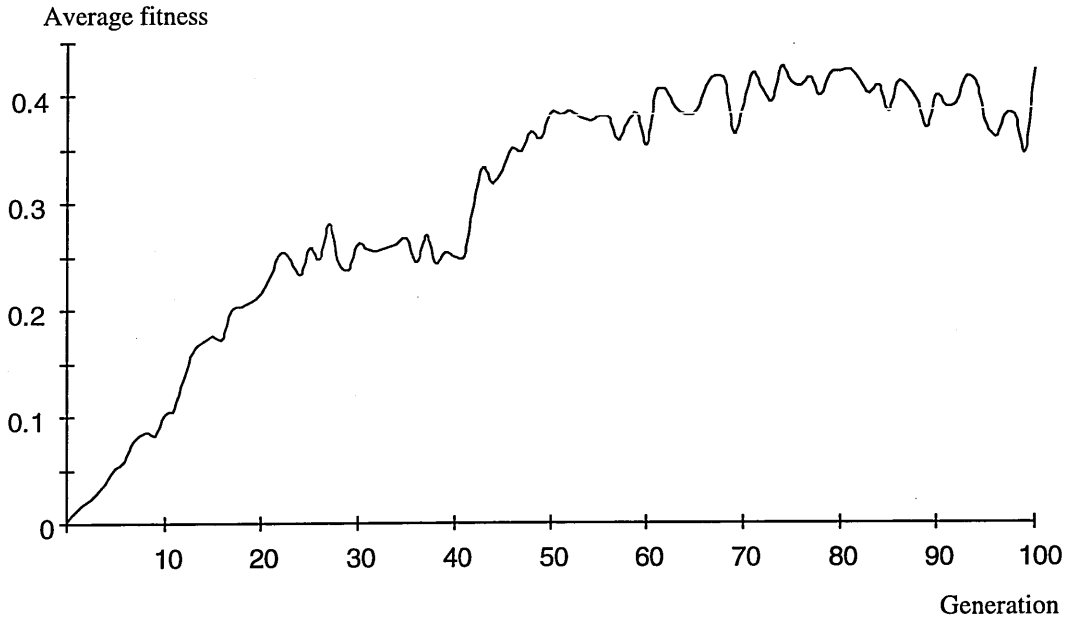


Figure 5.63. The average fitness against generation number

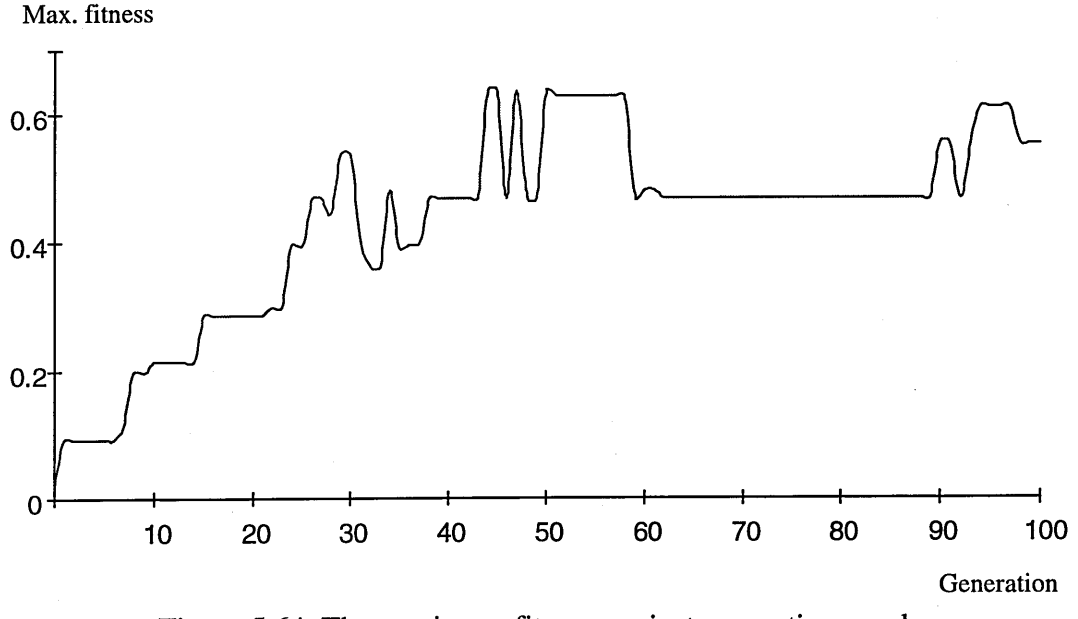


Figure 5.64. The maximum fitness against generation number.

Experiment 5.9.11:

Desired function	f(v)=0.45*v+0.85
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	320
Max. Generation	100

Fitness function:

$$f = \frac{|correlation_coef.|}{1 + \sum error_i^2}$$

The best individual of 100 generations was:

f(v)=SQRT((SQRT(0.460)+(v+0.000))).

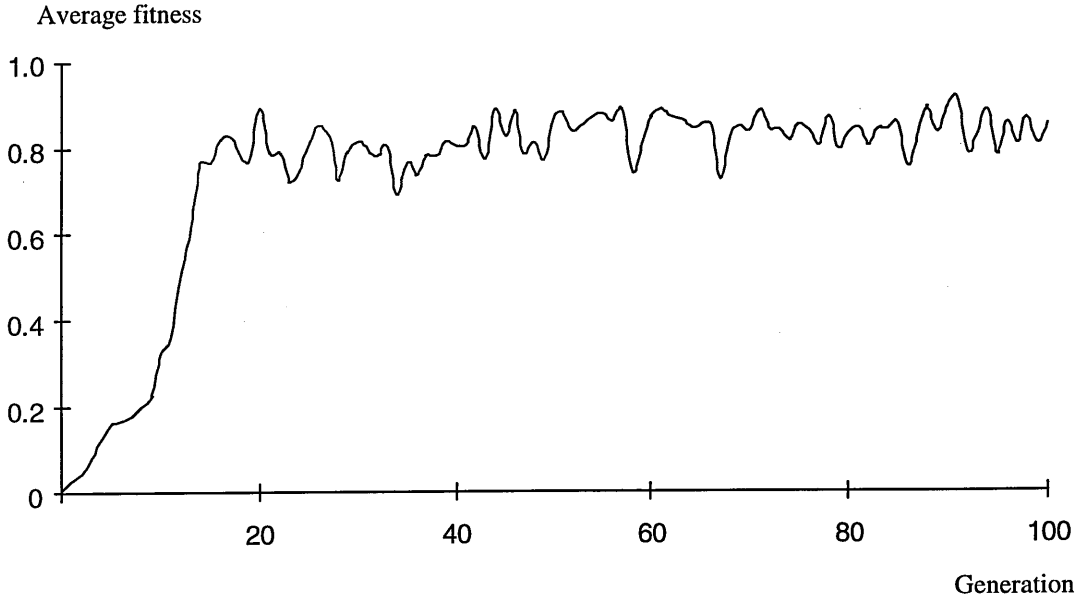


Figure 5.65. The average fitness against generation number

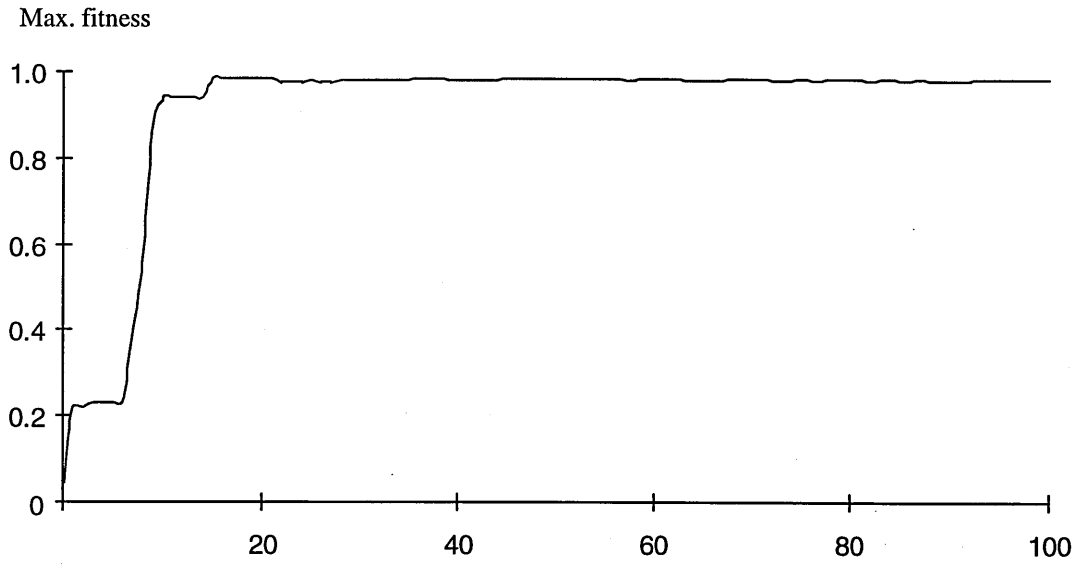


Figure 5.66. The maximum fitness against generation number.

• The Function *Sine*

Experiment 5.9.12:

Desired function	f(*v*)=0.43*sin(4**v***π+1.88)
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	480
Max. Generation	100

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

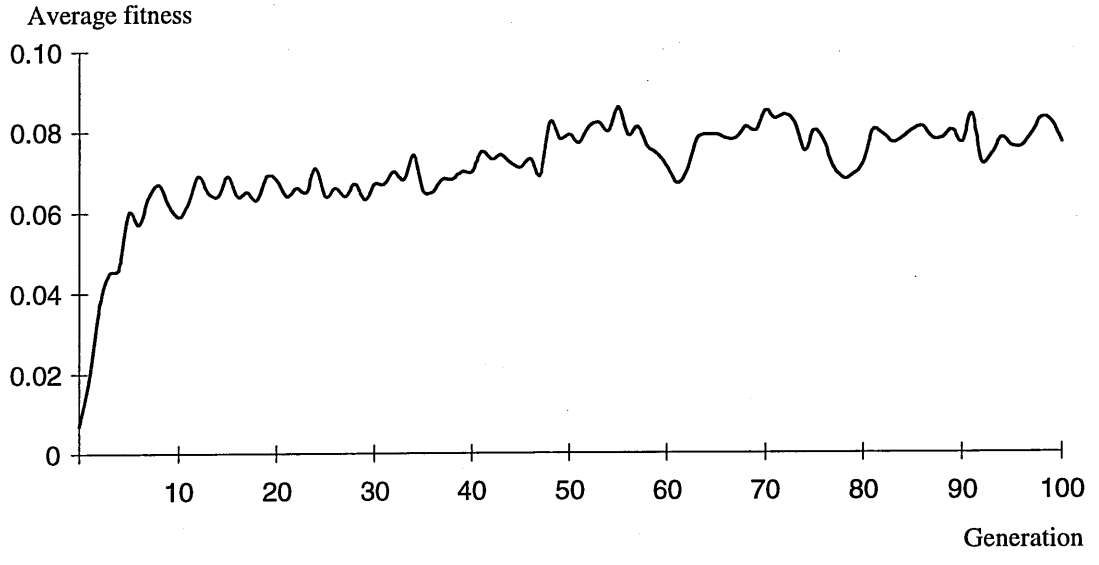


Figure 5.67. The average fitness against generation number

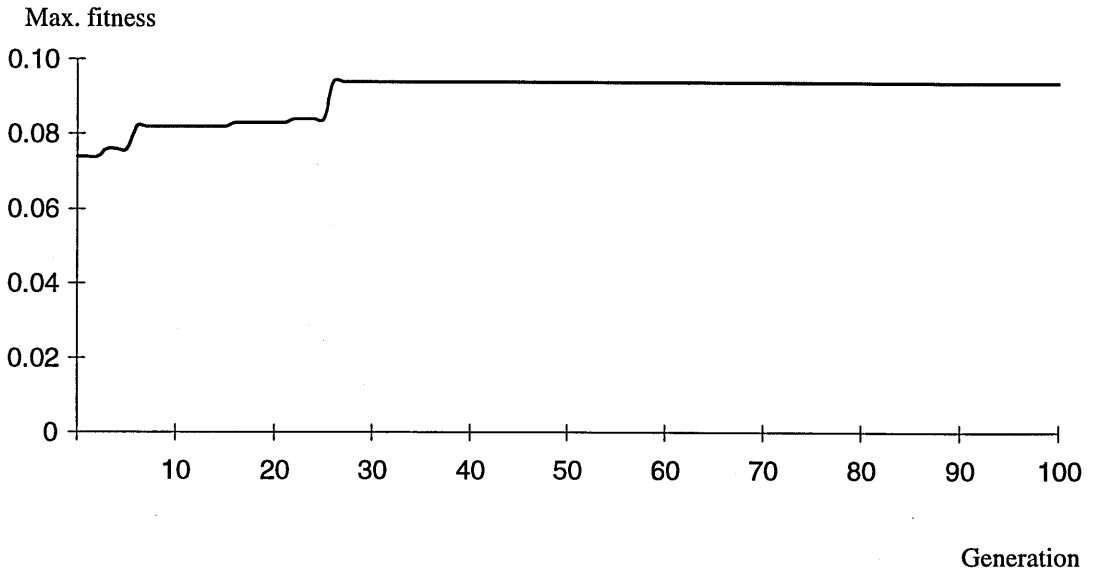


Figure 5.68. The maximum fitness against generation number.

Experiment 5.9.13:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	160
Max. Generation	100

Fitness function:

$f=|correlation_coef.|$

The best individual of 27 generations was:

f(v)=SQR(SIN(2.260)*SIN((v+SIN(2.260))*2*π)) with the fitness 0.999.

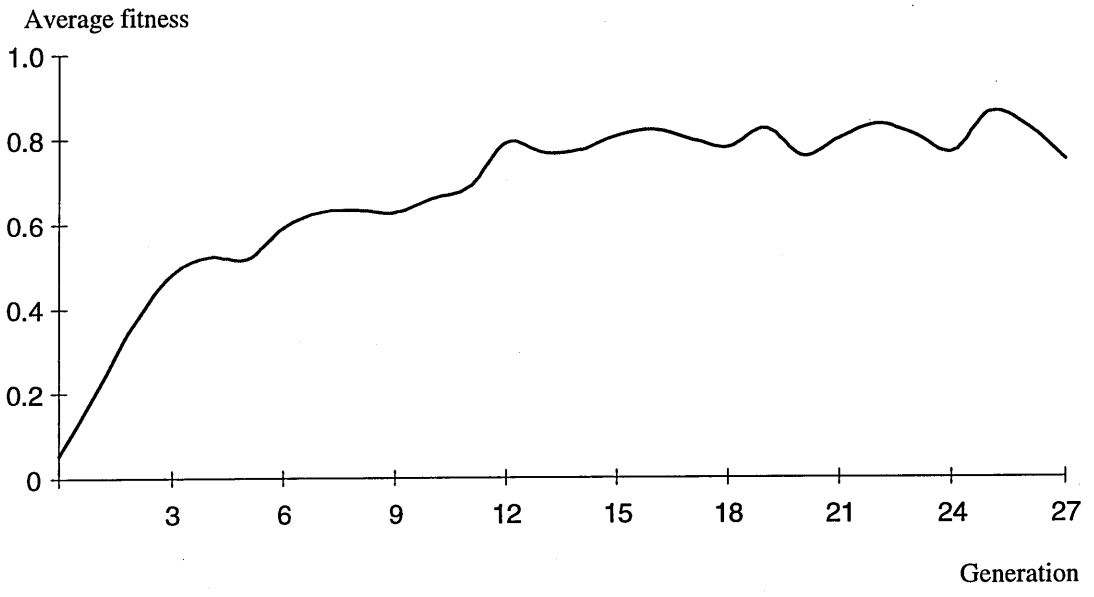


Figure 5.69. The average fitness against generation number

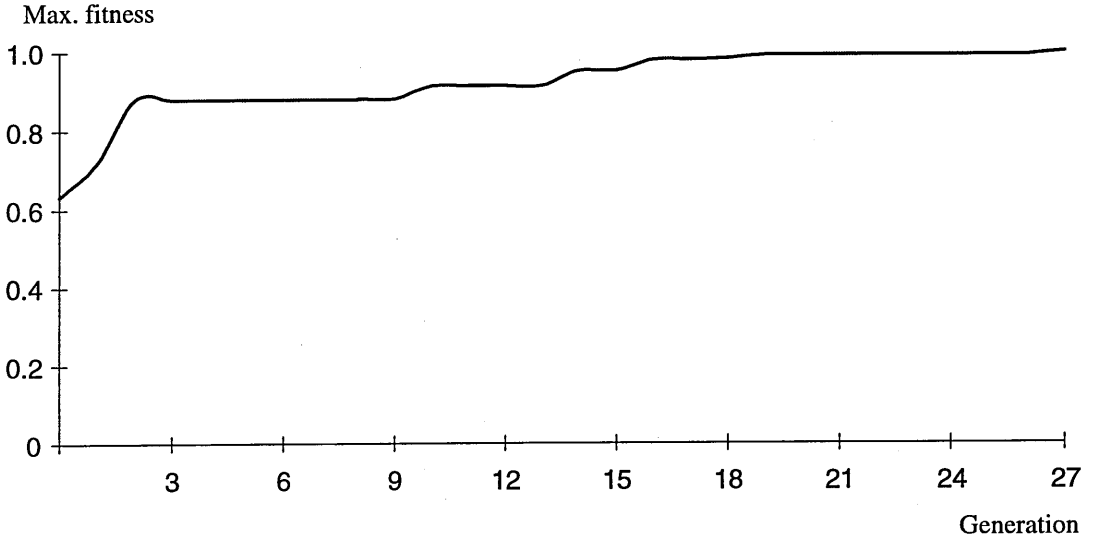


Figure 5.70. The maximum fitness against generation number.

Experiment 5.9.14:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	320
Max. Generation	100

Fitness function:

$f = |correlation_coef.|$

The best individual of 15 generations was:

f(v)=SQRT(SQR(SQR(TAN(1.070)*SIN(((v-2.050)+TAN(1.070))*2*π)
* 0.440))) with the fitness 0.999.

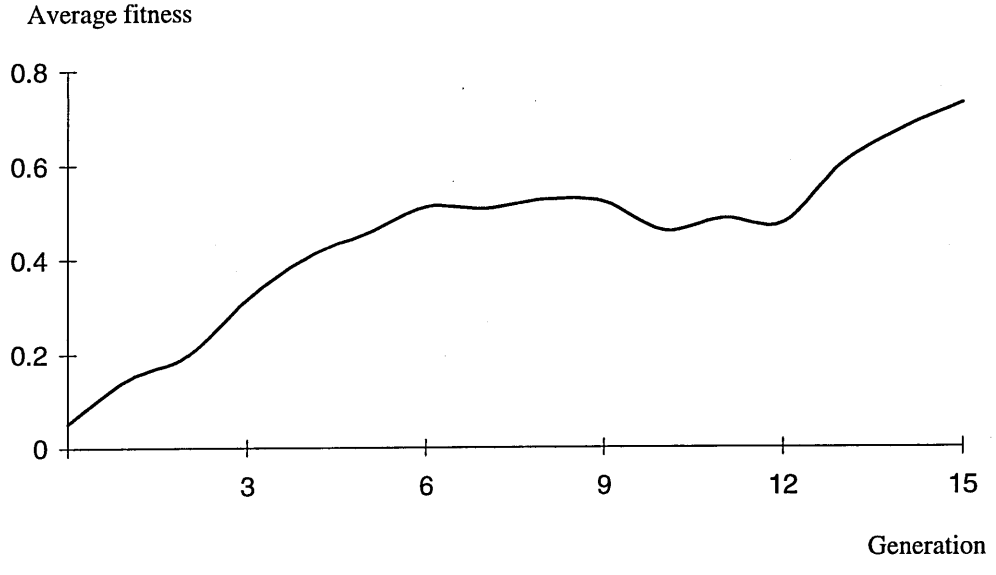


Figure 5.71. The average fitness against generation number

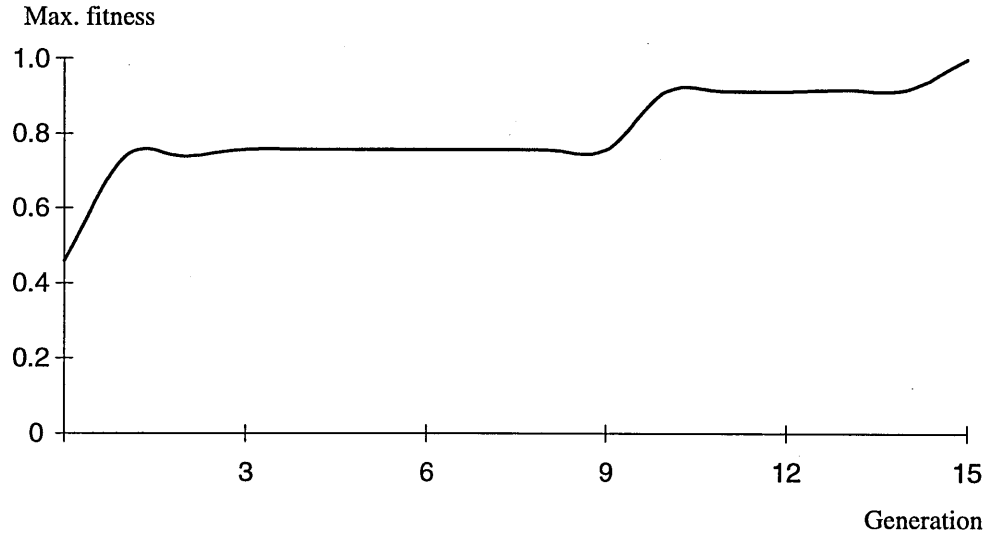


Figure 5.72. The maximum fitness against generation number.

Experiment 5.9.15:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	480
Max. Generation	100

Fitness function:

$f = |correlation_coef.|$

The best individual of 20 generations was:

f(v)=SIN((1.140+1.710))+(SIN((2.150*SQRT((v*v/0.030))-1.190))-0.960)

+ (2.040/2.300) with the fitness 0.999.

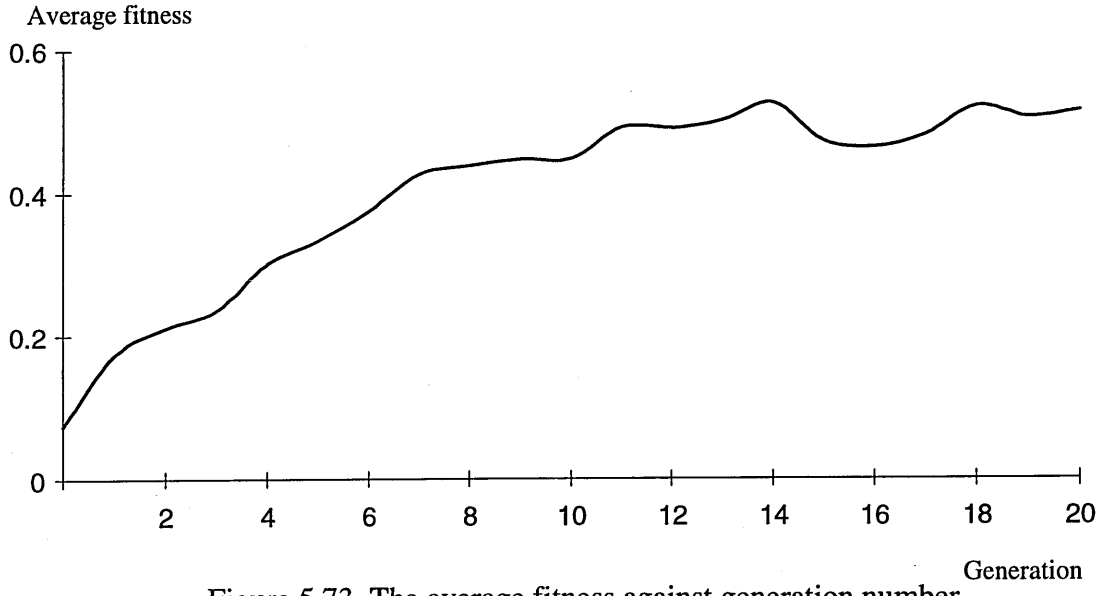


Figure 5.73. The average fitness against generation number

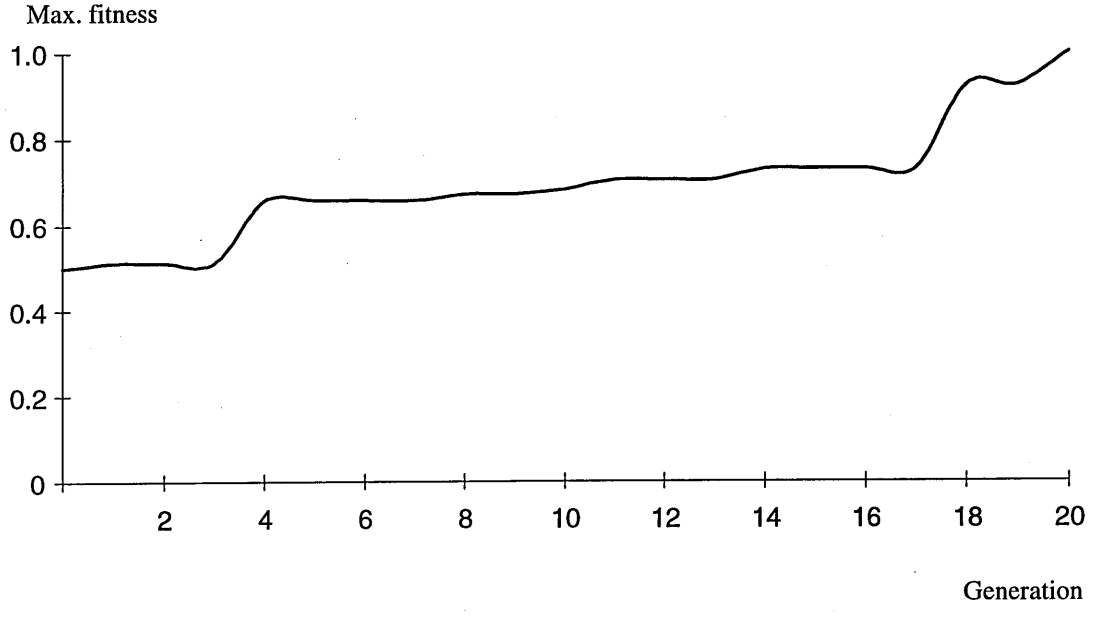


Figure 5.74. The maximum fitness against generation number.

• The fourth function.

Experiment 5.9.16:

Desired function	f(*v*)= (A+**n**)*Sin(((B+**n**)*v+(C+**n**))*2*π)*Cos((D+**n**)**v*)
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003
Length of Individual	640
Max. Generation	1000

Fitness function:

$f = |correlation_coef.|$

In this experiment, the fitness was the absolute correlation. The length of an individual was 640. The best fitness of the initial population was '0.587'. After performing 1000 generations, the model GP-1 found the following function with the fitness 0.952.

f(*v*)=SIN(TAN(1.100))+SIN(TAN(SQRT(1.700*SIN((*v*+1.700)*2*π)
*(0.160/1.730))))+(*v**SIN((0.800*1.810+*v*)*2*π)+TAN((0.920-*v*)))

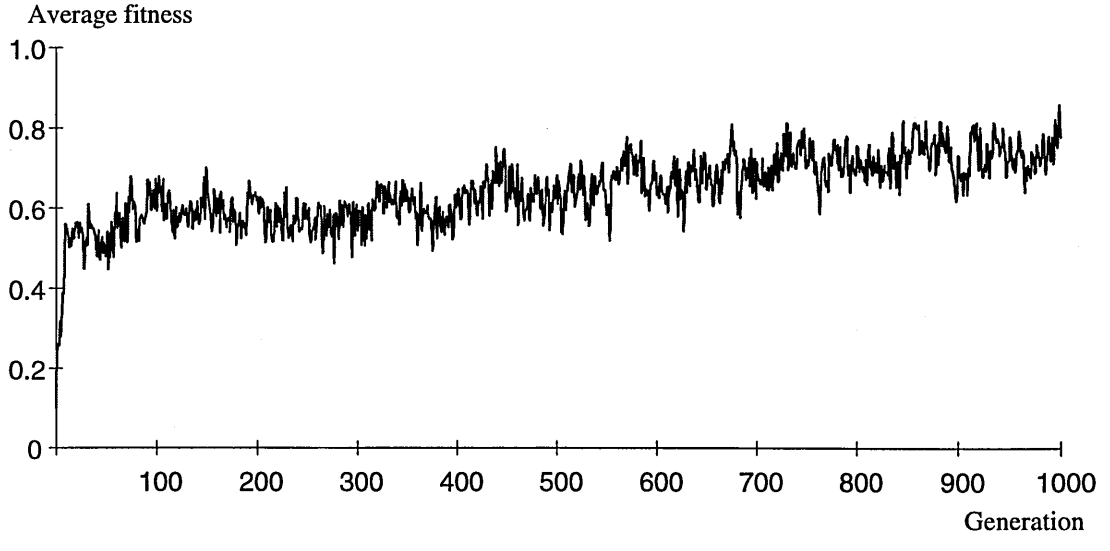


Figure 5.75. The average fitness against generation number

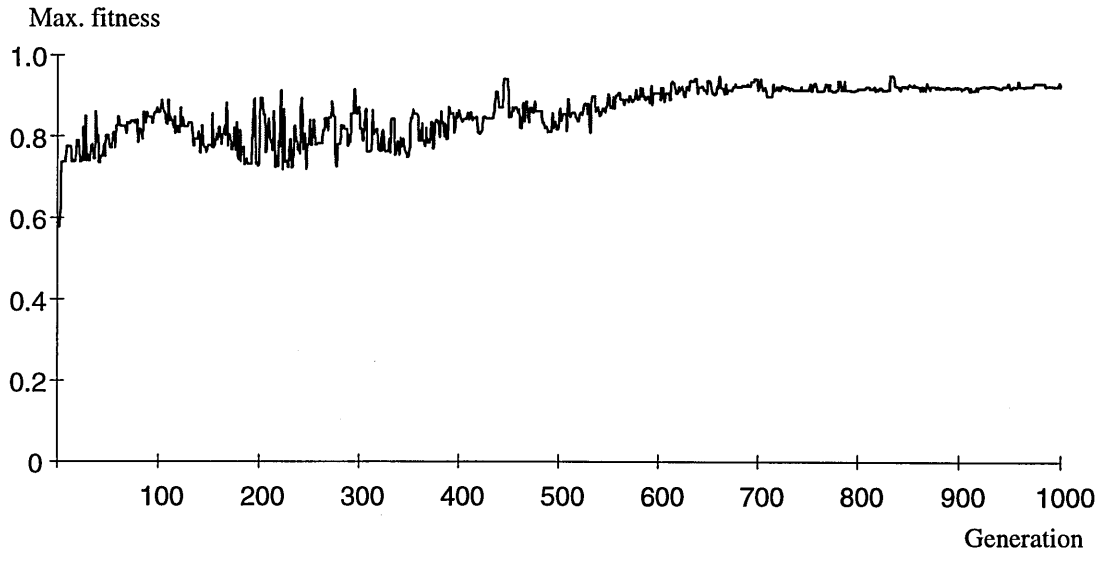


Figure 5.76. The maximum fitness against generation number.

Because the correlation was -0.952, the output of GP-1 was multiplied by -1.

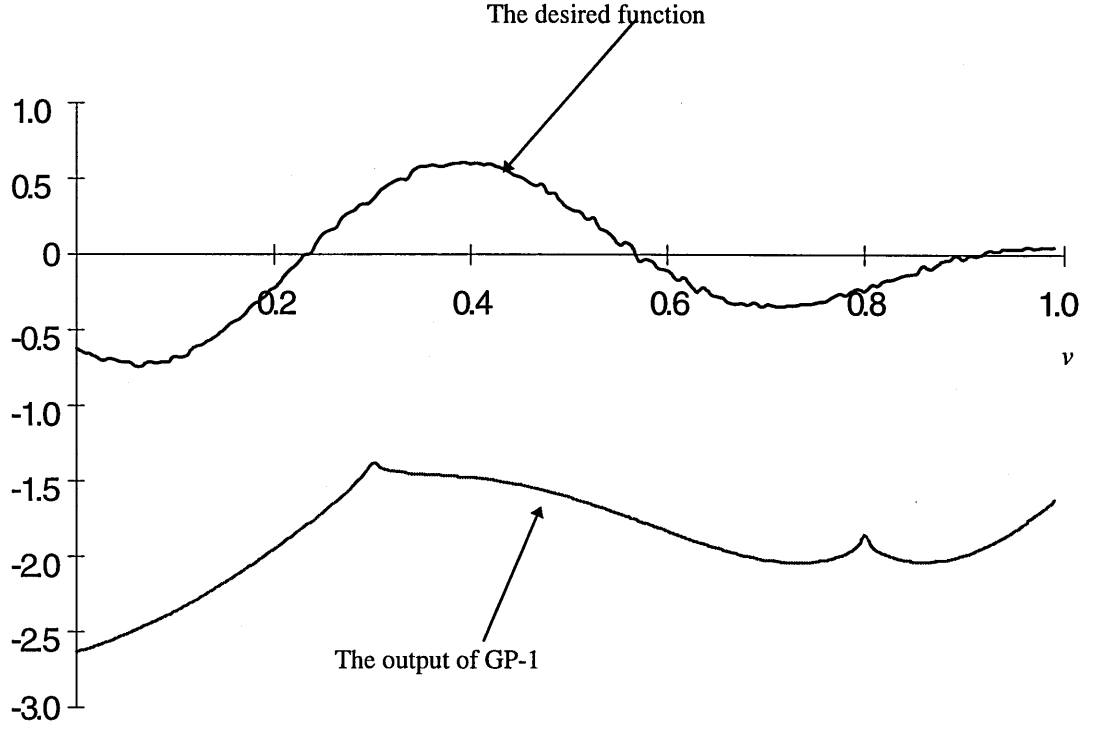


Figure 5.77. The desired function and the best of the 1000 generations.

5.10. TESTING The MODEL GP-2

Experiment 5.10.1:

Desired function	f(v)=0.243
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003

Fitness function:

$$f=\frac{1}{1+\sum error_i^2}$$

The best individual of the initial population : f(*v*)=*v*0.376*

The best individual of the last generation (740th): f(*v*)=*0.241*

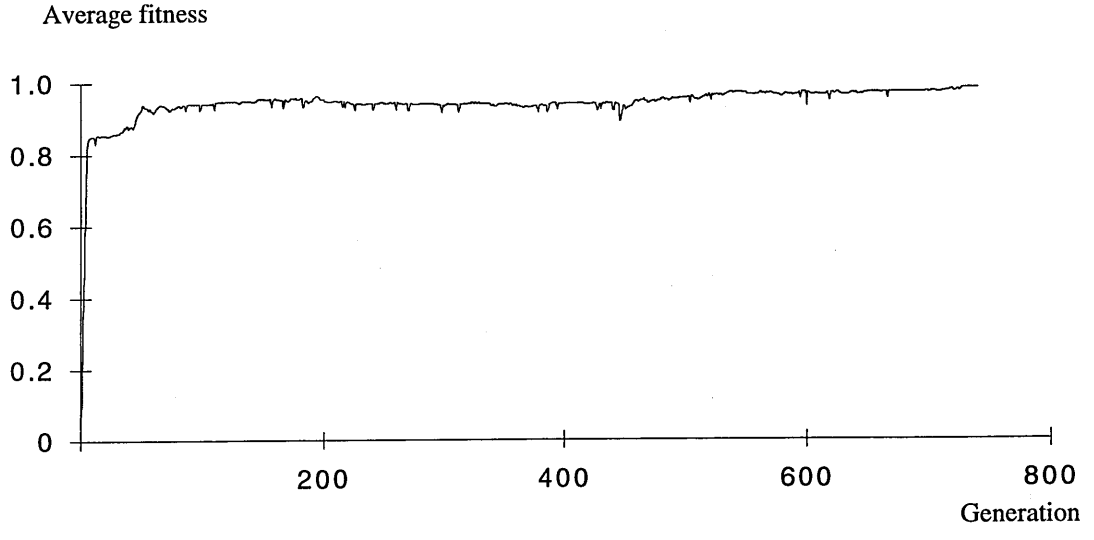


Figure 5.78. The average fitness against generation number.

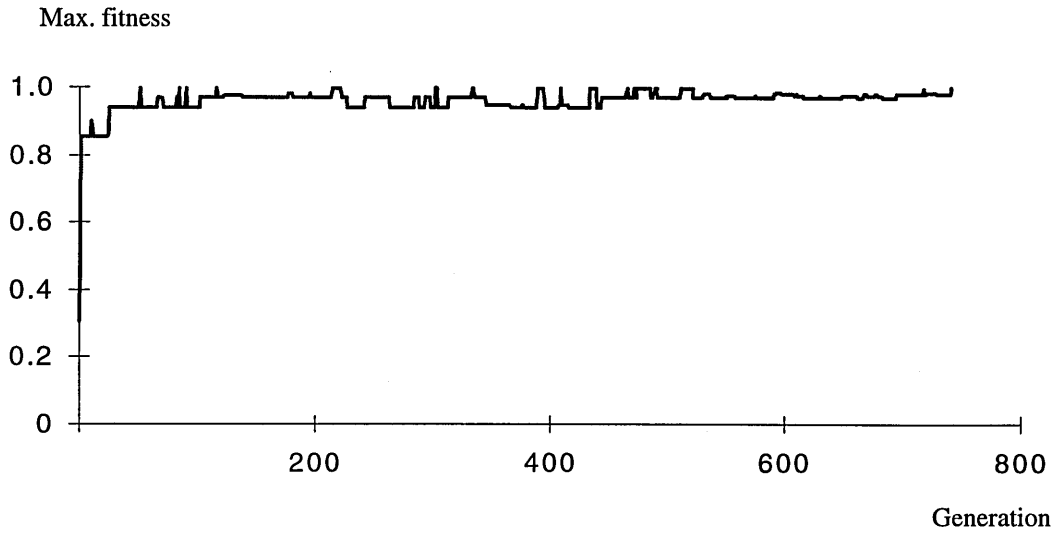


Figure 5.79. The maximum fitness against generation number.

Experiment 5.10.2:

Desired function	f(v)=0.243
Population size	100
Crossover Probability	0.750
Mutation Probability	0.003

Fitness function:

$$f = \frac{1}{1 + \sum error_i^2}$$

Generation No	Average Fitness	Max. Fitness	The best individual
0	0.931	0.931	sin(0.223)
1	0.982	0.982	0.186/sqrt(abs(-0.534))
2	0.983	0.983	sqrt(abs(-0.290))*(0.878-0.406)
3	0.995	0.995	sin(0.186/sqrt(abs(-0.534)))/1.059
4	0.997	0.997	sqrt(abs(sin(0.186)))*0.555
5	0.983	0.983	sqrt(abs(-0.290))*(0.878-0.406)
6	0.997	0.997	sqrt(abs(sin(0.186)))*0.555
7	0.997	0.997	sqrt(abs(sin(0.186)))*0.555
8	0.997	0.997	sqrt(abs(sin(0.186)))*0.555
9	0.960	0.960	sqrt(abs(sin(sin(0.223))))*0.555
10	0.996	0.996	0.248
11	0.949	0.949	sqrt(abs(0.223))*0.555
12	0.941	0.941	0.223
13	0.999	0.999	sin(0.248)

Table 5.7. The best individuals in symbolic form.

Experiment 5.10.3:

Desired function	f(v)=0.45*v+0.85
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003

Fitness function:

$$f = \frac{1}{1 + \sum error_i^2}$$

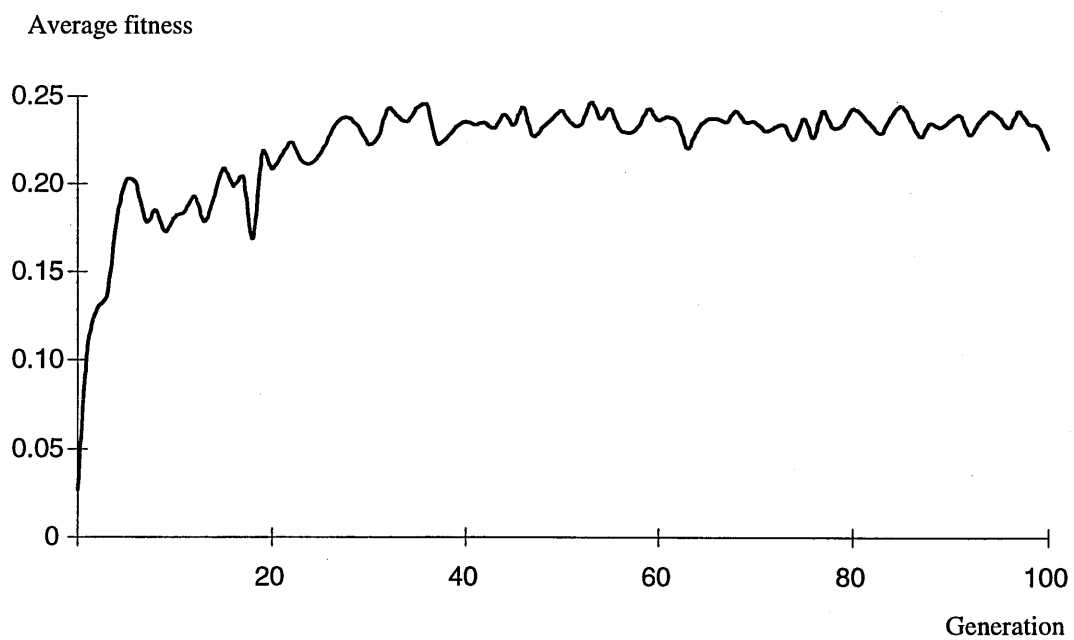


Figure 5.80. The average fitness against generation number.

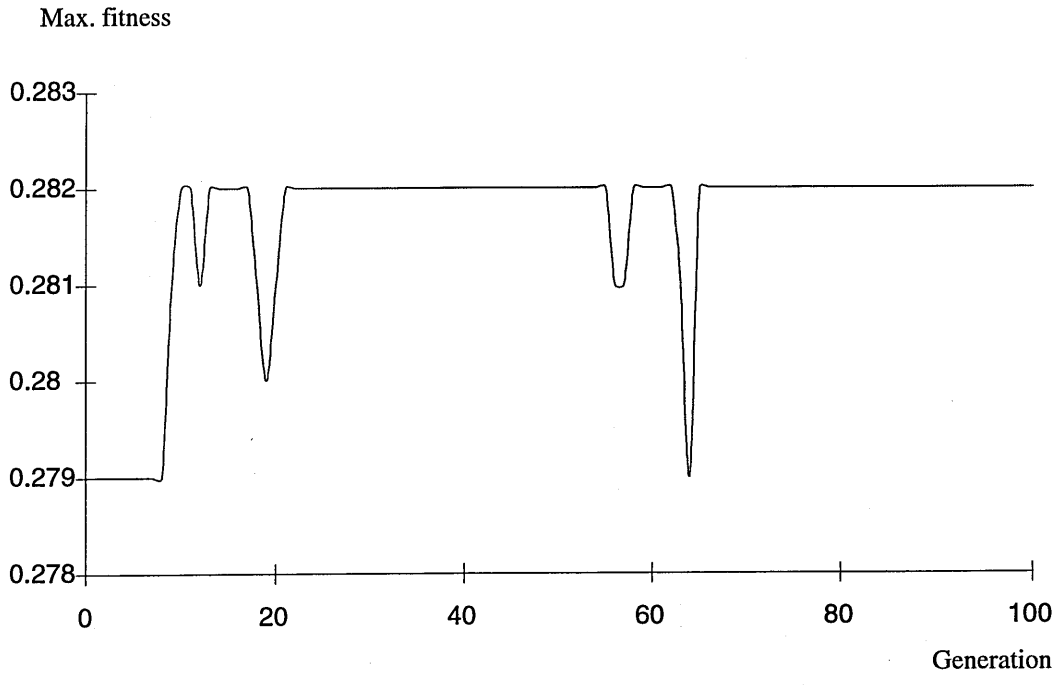


Figure 5.81. The maximum fitness against generation number.

Experiment 5.10.4&5&6:

Desired function	f(v)=0.45*v+0.85
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003

Fitness function:
$f=|correlation_coef.|$

The GP-2 found the best in the initial population.

Experiment no	Function
4	v-(-0.825)
5	-0.417+v
6	v

Table 5.8. The best individuals in symbolic form.

Experiment 5.10.7:

Desired function	f(*v*)=0.45**v*+0.85
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003

Fitness function:

$$f = \frac{|correlation_coef.|}{1 + \sum error_i^2}$$

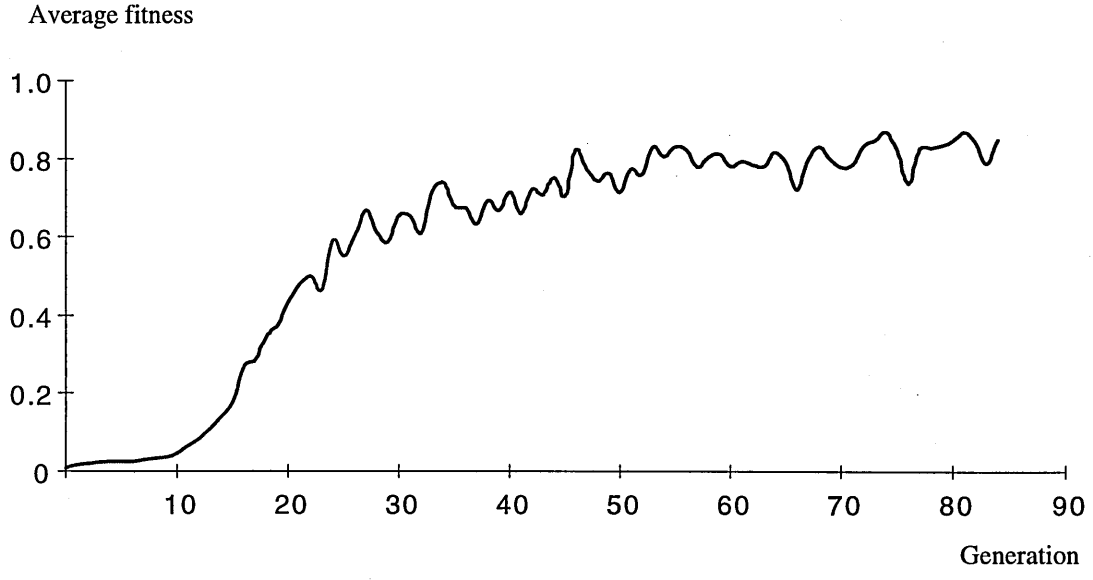


Figure 5.82. The average fitness against generation number.

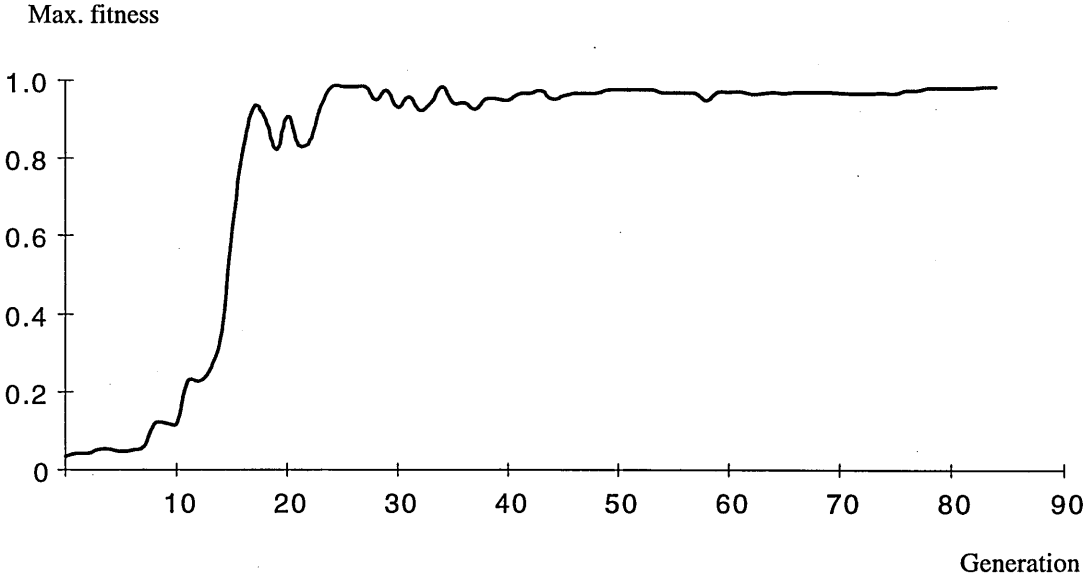


Figure 5.83. The maximum fitness against generation number.

Experiment 5.10.8:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003

Fitness function:

$$f = \frac{1}{1 + \sum error_i^2}$$

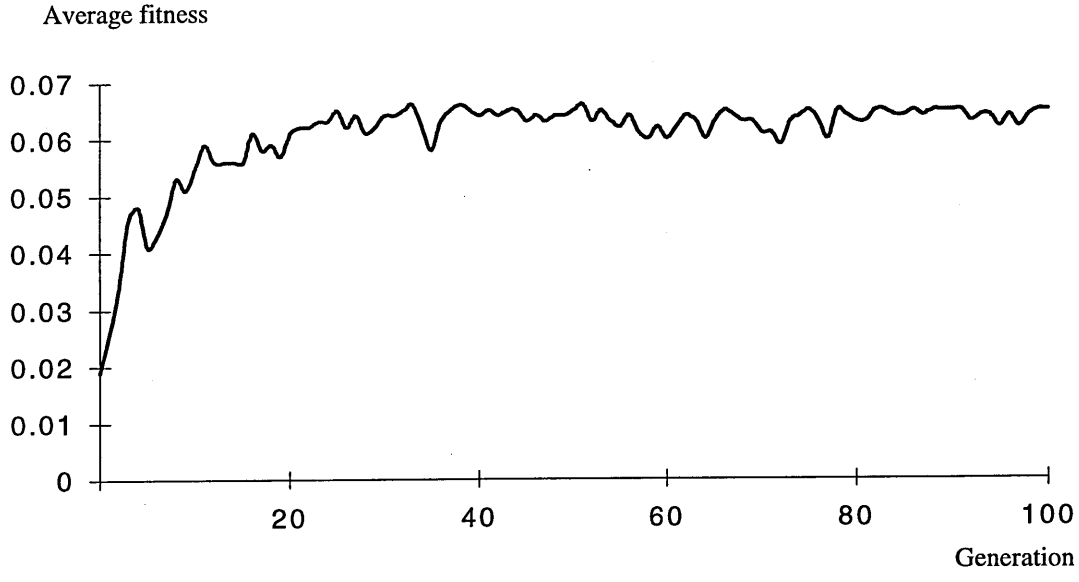


Figure 5.84. The average fitness against generation number.

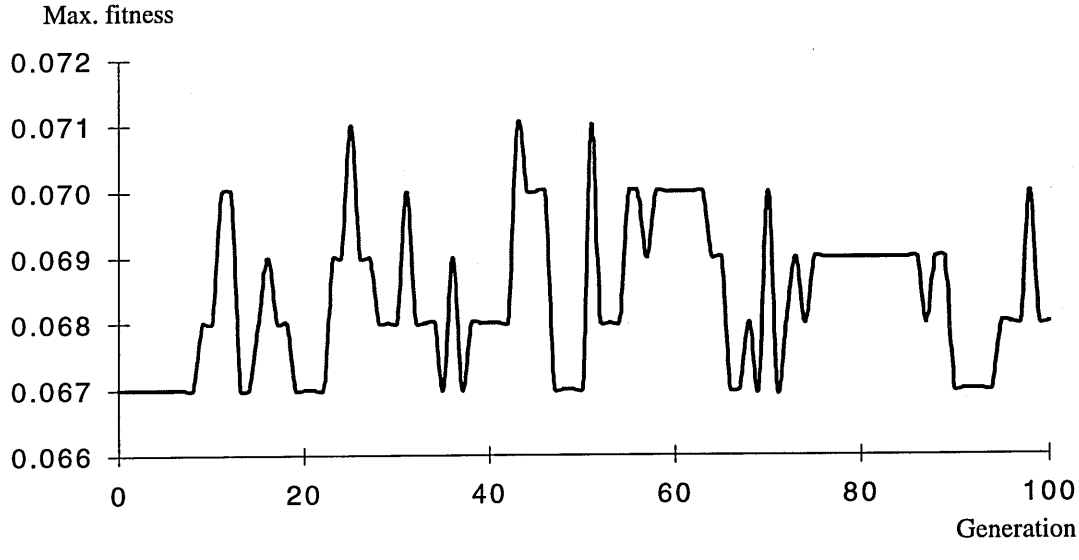


Figure 5.85. The maximum fitness against generation number.

Experiment 5.10.9:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	50
Crossover Probability	0.750
Mutation Probability	0.003

Fitness function:

$f = |correlation_coef.|$

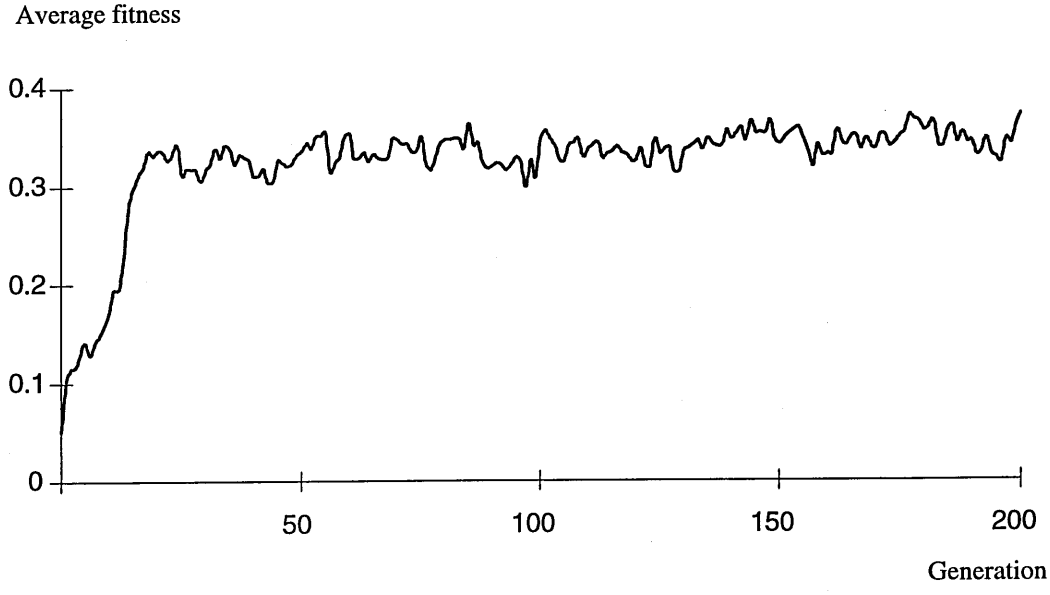


Figure 5.86. The average fitness against generation number.

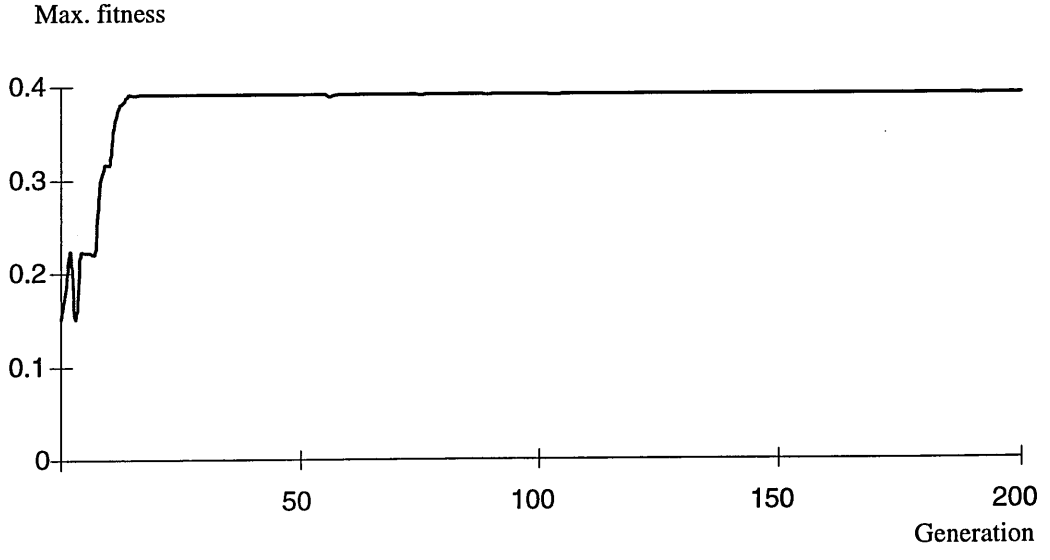


Figure 5.87. The maximum fitness against generation number.

Experiment 5.10.10:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	500
Crossover Probability	0.750
Mutation Probability	0.003

Fitness function:
$f=|correlation_coef.|$

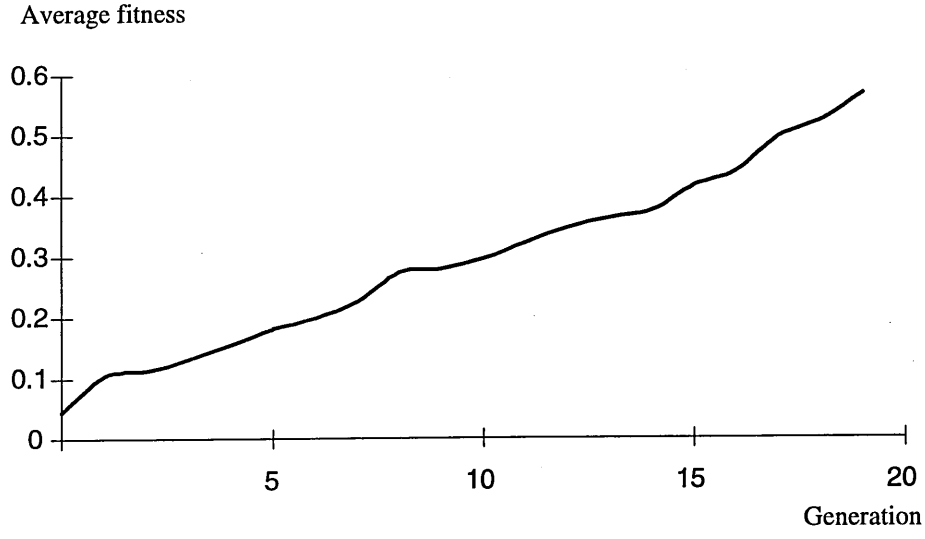


Figure 5.89. Average fitness vs. generation number.

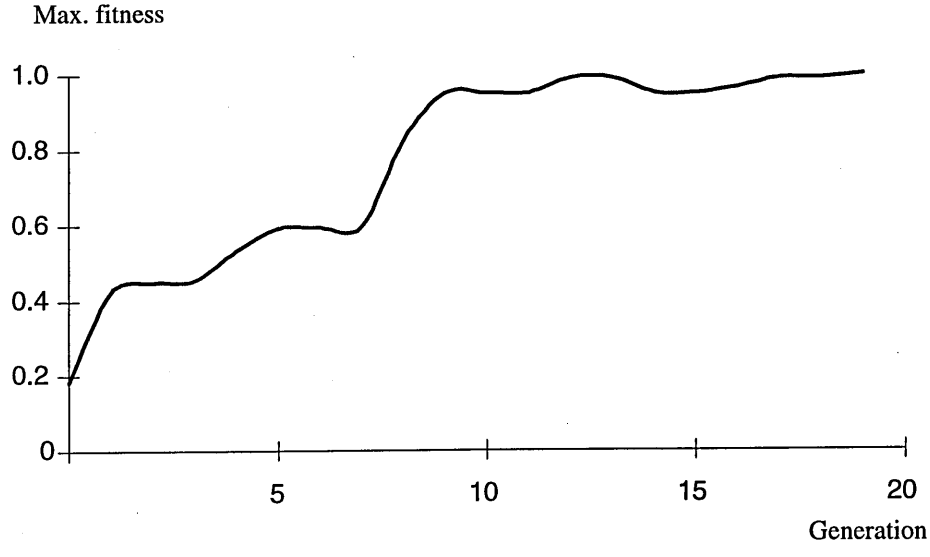


Figure 5.90. Maximum fitness vs. generation number.

Generation	Average Fitness	Max. Fitness	Function
0	0.045	0.185	sin(1.212*2*π+v)
1	0.105	0.425	sin((v*v)*2*π+v)
2	0.113	0.449	sin((v*v)*2*π+sqr(v))
3	0.132	0.454	sin((v*(v+0.251))*2*π+sqr(abs(v)))
4	0.155	0.534	(sin((v*v)*2*π+sqr(abs(v))))-(0.486-v)
5	0.181	0.593	sqr(abs(v+(sin((v*v)*2*π+v))))
6	0.199	0.595	(v*(sin((v*v)*2*π+v)))+v
7	0.226	0.596	sin((v*(1.120*v))*2*π+(v*v))
8	0.274	0.826	sqr(abs(sin(v*2*π+sqr(v))))
9	0.280	0.951	sqr(abs(sin(v*2*π+(v-v))))
10	0.297	0.951	sqr(abs(sin(v*2*π+(v-v))))
11	0.324	0.951	sqr(sin(v*2*π+(v-v)))
12	0.347	0.991	sqr(abs(sin(v*2*π+(v-(v*v)))))
13	0.363	0.991	sqr(abs(sin(v*2*π+(v-(v*v)))))
14	0.375	0.951	sqr(abs(sin(v*2*π+(v-v))))
15	0.417	0.951	sqr(abs(sin(v*2*π+(v-v))))
16	0.44	0.966	sqr(abs(sin(v*2*π+(v*0.055))))
17	0.496	0.991	sqr(abs(sin(v*2*π+(v-(v*v)))))
18	0.524	0.991	sqr(abs(sin(v*2*π+(v-(v*v)))))
19	0.568	1.000	sqr(abs(sin(v*2*π+sqr(0.377))))

Table 5.9. The best individuals in symbolic form.

Experiment 5.10.11:

Desired function	f(v)=0.43*sin(4*v*π+1.88)
Population size	500
Crossover Probability	0.750
Mutation Probability	0.003

Fitness function:

$$f = \frac{|correlation_coef.|}{1 + \sum error_i^2}$$

The best individual of the initial population: f(v)= 0.016*v

The best individual of the last generation:

f(v) =sqr(sqr(sqr(sqr((v-sqr(sqr(v)))*0.963))))*((-0.387+sqr(v))
*sqr(0.611-(sqr((sqr(v)-(v*v))*sqr(v))*(sqr(0.606*sqr(sqr(v)))*v))))

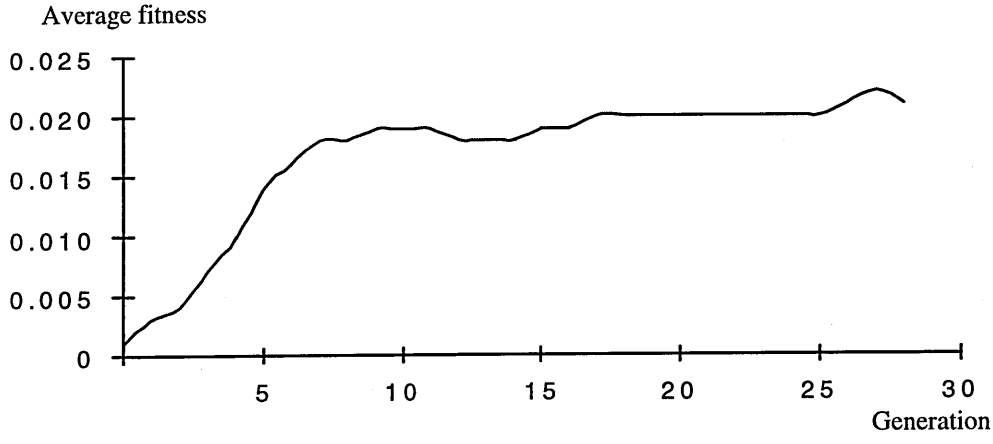


Figure 5.91. Average fitness vs. generation number.

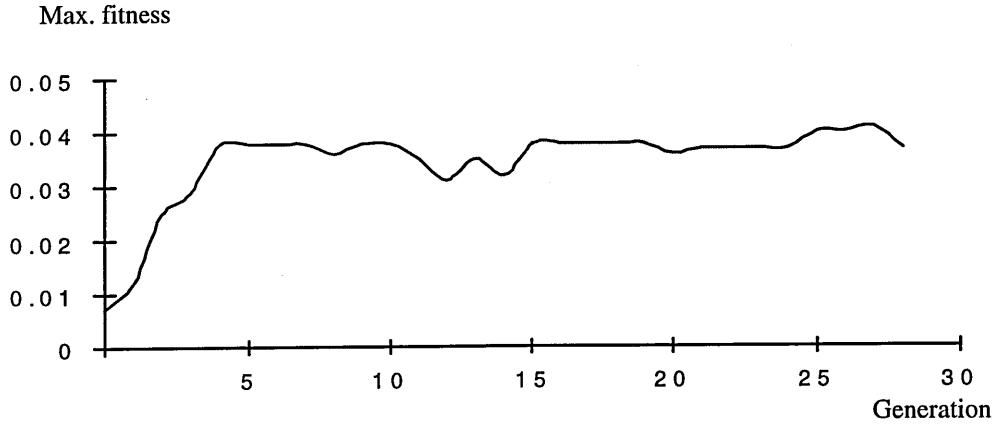


Figure 5.92. Maximum fitness vs. generation number.

Experiment 5.10.12:

Desired function	f(v)= (A+**n**)*Sin(((B+**n**)*v+(C+**n**))*2*π)*Cos((D+**n**)*v)
Population size	500
Crossover Probability	0.750
Mutation Probability	0.003

Fitness function:

$f = |correlation_coef.|$

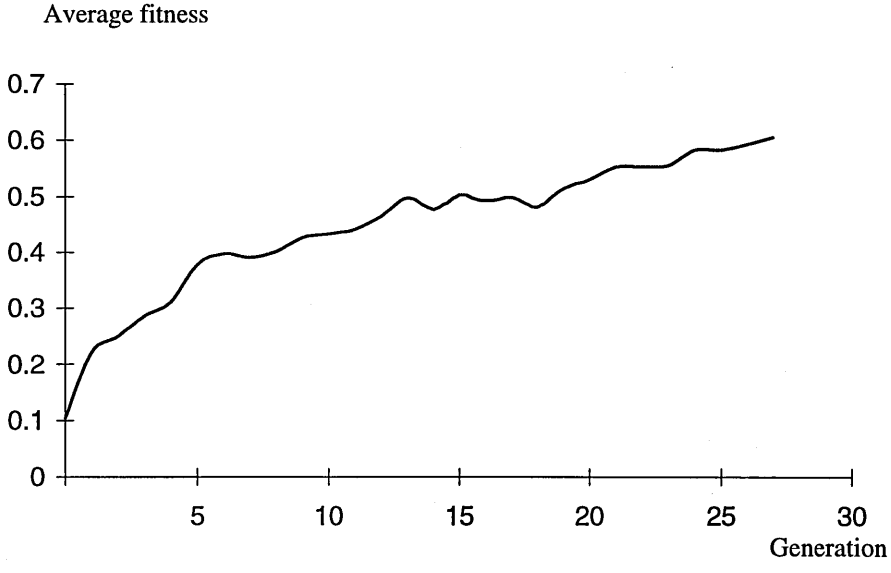


Figure 5.93. Average fitness vs. generation number.

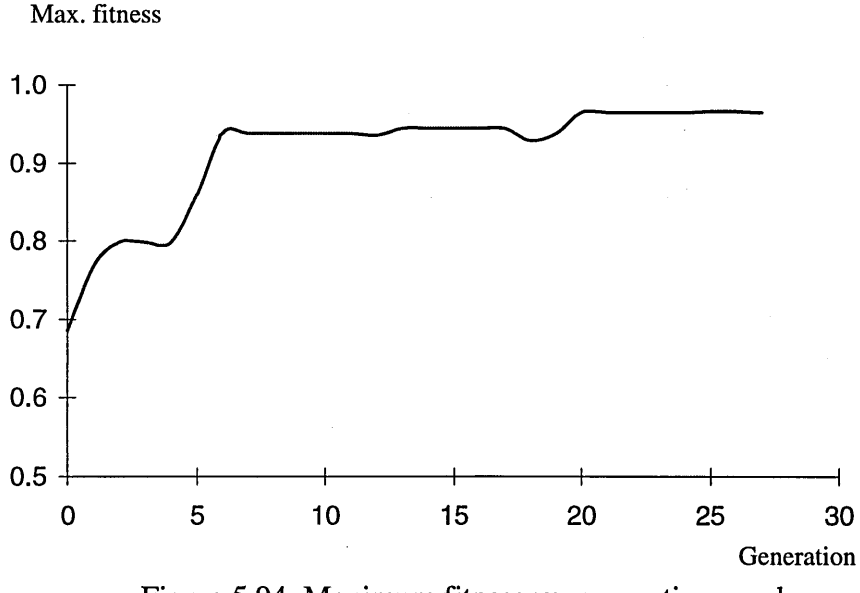


Figure 5.94. Maximum fitness vs. generation number.

Generation	Average Fitness	Max. Fitness	Function
0	0.105	0.686	sin(v*2*π+-1.340)
1	0.222	0.768	sin((v*1.486)*2*π+v)
2	0.252	0.798	sin(v*2*π+(v-1.406))
3	0.287	0.798	sin(v*2*π+(v-1.406))
4	0.312	0.798	sin(v*2*π+(v-1.406))
5	0.378	0.859	sin((v*1.512)*2*π+sqrt(abs(sin(v))))
6	0.397	0.938	sin((v*1.512)*2*π+sqrt(abs(sin(sin(0.657*2*π+v)))))
7	0.39	0.938	sin((v*1.512)*2*π+sqrt(abs(sin(sin(0.657*2*π+v)))))
8	0.426	0.938	sin((v*1.512)*2*π+sqrt(abs(sin(sin(0.657*2*π+v)))))
9	0.433	0.938	sin((v*1.512)*2*π+sqrt(abs(sin(sin(0.657*2*π+v)))))
10	0.433	0.938	sin((v*1.512)*2*π+sqrt(abs(sin(sin(0.657*2*π+v)))))
11	0.44	0.938	sin((v*1.512)*2*π+sqrt(abs(sin(sin(0.657*2*π+v)))))
12	0.463	0.936	(sin(v*2*π+(sin((v*-0.356)*2*π+-1.340))))+sin(v)
13	0.497	0.945	sin(sqrt(abs(v))*2*π+sqrt(abs((sin(sqrt(abs(v))*2*π+ (sin((v*-0.589)*2*π+-1.340))))+-0.818)))
14	0.477	0.945	sin(sqrt(abs(v))*2*π+sqrt(abs((sin(sqrt(abs(v))*2*π+ (sin((v*-0.589)*2*π+-1.340))))+-0.818)))
15	0.502	0.945	sin(sqrt(abs(v))*2*π+sqrt(abs((sin(sqrt(abs(v))*2*π+ (sin((v*-0.589)*2*π+-1.340))))+-0.818)))
16	0.492	0.945	sin(sqrt(abs(v))*2*π+sqrt(abs((sin(sqrt(abs(v))*2*π+ (sin((v*-0.589)*2*π+-1.340))))+-0.818)))
17	0.499	0.945	sin(sqrt(abs(v))*2*π+sqrt(abs((sin(sqrt(abs(v))*2*π+ (sin((v*-0.589)*2*π+-1.340))))+-0.818)))
18	0.482	0.929	sin((v*1.512)*2*π+sqrt(abs(sqrt(abs(sin(v*2*π+ (sin(v*2*π+-1.340))))))))
19	0.514	0.938	sin((v*1.512)*2*π+sqrt(abs(-0.818)))
20	0.53	0.965	((sin(v*2*π+(sin((0.093*-1.340)*2*π+ (sin(-1.340*2*π+v))))))*v)-(sin(v*2*π+(v-1.432)))
21	0.553	0.965	((sin(v*2*π+(sin((0.093*-1.340)*2*π+ (sin(-1.340*2*π+v))))))*v)-(sin(v*2*π+(v-1.432)))
22	0.553	0.965	((sin(v*2*π+(sin((0.093*-1.340)*2*π+ (sin(-1.340*2*π+v))))))*v)-(sin(v*2*π+(v-1.432)))
23	0.556	0.965	((sin(v*2*π+(sin((0.093*-1.340)*2*π+ (sin(-1.340*2*π+v))))))*v)-(sin(v*2*π+(v-1.432)))
24	0.583	0.965	((sin(v*2*π+(sin((0.093*-1.340)*2*π+ (sin(-1.340*2*π+v))))))*v)-(sin(v*2*π+(v-1.432)))
25	0.583	0.967	((sin(v*2*π+(sin((0.093*-1.334)*2*π+(sin(-1.340*2*π -0.624))))))*v)-(sin(v*2*π+(v-1.432)))
26	0.593	0.967	((sin(v*2*π+(sin((0.093*-1.334)*2*π+(sin(- 1.340*2*π+-0.624))))))*v)-(sin(v*2*π+(v-1.432)))
27	0.606	0.965	((sin(v*2*π+(sin((0.093*-1.333)*2*π+ (sin(-1.340*2*π+v))))))*v)-(sin(v*2*π+(v-1.432)))

Table 5.10. The best individuals in symbolic form.

The best output of the GP-2 was:

f(v)=((sin(v*2*π+(sin((0.093*-1.334)*2*π+(sin(-1.340*2*π+-0.624))))))*v)
-(sin(v*2*π+(v-1.432)))

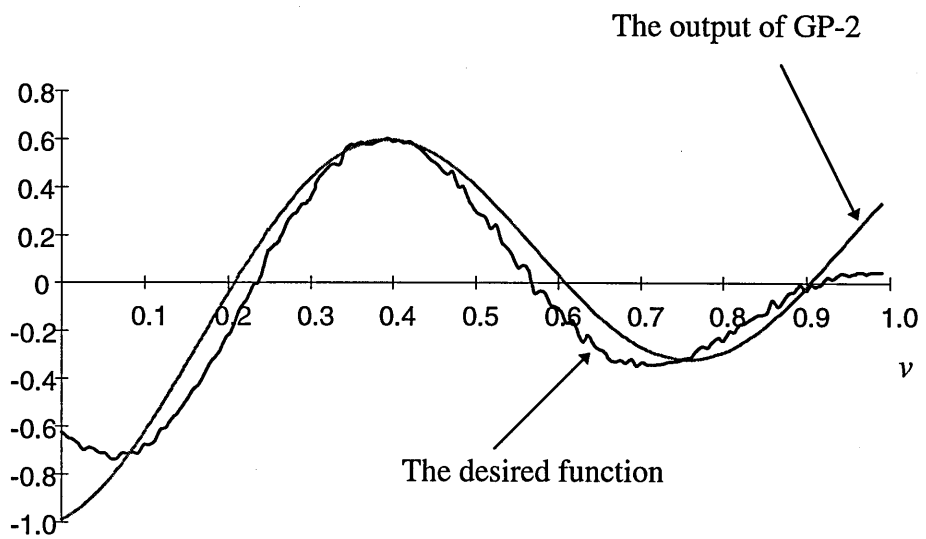


Figure 5.95. The desired function and the best output of the GP-2.

CHAPTER 6

SYMBOLIC REGRESSION for STRAIN on a HELICOPTER's MAIN ROTOR

6.1. INTRODUCTION

The cost of maintenance of helicopters might be drastically reduced if we could somehow measure the strains that their critical parts suffer during use. Unfortunately, these critical parts tend to be rotating and very difficult to instrument, except at great expense and even then only for development purposes (Vella et al 1996). The aim of this part of the research was to use genetic algorithms and genetic programming to search for functional forms of approximations of the strain data. This function can be used to predict strains on the moving parts of helicopter rotors using data collected at more accessible fixed positions.

6.2. PROBLEM DEFINITION

In a helicopter, the main and tail rotors, the engine and transmission, a number of other rotating components, and the airframe itself cause vibrations (Lison 1994). In general, helicopters are subjected to much more vibratory loading conditions than fixed-wing aircraft (Crawford et al 1991).

To avoid technical problems and incidents, preventive maintenance is used to satisfy the required specifications of the vibrations. It is also important to safeguard the value of investment, operate within the law, inspire confidence in flight crew and customers, and provide safe, reliable and airworthy aircraft (Bateman 1982).

For a particular component or system, primary maintenance can be defined as one of the following (Wendlandt 1990):

• Hard time: It periodically overhauls the component or the system or removes it after certain period of usage based on operating experience or tests as appropriate.

• On condition: As a preventive maintenance, it periodically inspects the component or the system against some appropriate physical limit to determine whether it can continue in service. These limits can be adjusted based on operating experience or tests as appropriate.

• Condition monitoring: This is accomplished by means of data collection and analysis by which an operator obtains information from the whole population of a system, or item in service.

Beside many benefits of preventive maintenance, the disadvantages are almost entirely concerned with the cost of the activity (Bateman 1982).

As a preventive maintenance method, helicopter health and usage monitoring is an attempt to improve reliability and maintainability, and to reduce operating costs. It seeks to identify changes or trends in monitored areas to provide advanced warning of component degradation towards failure, allowing the component to be changed before the failure occurs (Hill 1994).

In the UK military field, usage monitoring for helicopters is defined by the Ministry of Defence (MOD) Defence Standard (MOD-PEDS 1989):

"*Usage monitoring is a process which assesses the life consumption of life-limited components, systems and structures by monitoring actual damage exposure (e.g. due to combinations of loads, speeds, temperatures, etc.). Allied to this is recording of information relating to exceedances of placarded limits,*

whether in emergency conditions or normal operations. The activity relates primarily to components substantiated on a safe-life basis."

The civilian definition of usage monitoring is provided by the Helicopter Health Monitoring Advisory Group (CAA 1993) :

"*A process which assesses the life consumption of critical components, systems, and structures by monitoring actual damage exposure.*"

Helicopter usage monitoring can take many forms that vary widely in complexity. They range from simple automated timing systems, recording, for example, flying hours and engine running hours, to complex systems which monitor the structural usage or fatigue consumption of airframe and rotor components. Whilst simpler systems are already in service and contributing towards reduced running costs and improved availability, structural usage monitoring has yet to achieve operational maturity (Hill 1994).

Usage monitoring provides the benefit of utilising actual aircraft usage data rather than relying on an assumed usage spectrum. Fatigue life assessments based upon an assumed usage spectrum must include the worst case scenario and therefore are inherently conservative. Measuring or otherwise calculating the actual "damage" or fatigue felt by particular components would reduce the high degree of conservatism currently required in fatigue life assessment (Lison 1994).

To practically measure the component loads in the airframe and rotor system practically requires complex slip ring and strain gauge instrumentation. However, the instrument itself is also prone to damaging influences. Hence, this method is much problematic particularly for rotating components. In

addition to this, it costs too much because of such instrumentation. Therefore recent work has been focused around using fixed system parameters for predicting component loads (Hill et al 1995, Vella et al 1996, Lison 1994).

6.3. EVOLUTIONARY SEARCH ALGORITHMS and HELICOPTER HEALTH and USAGE MONITORING.

The aims of this part of the study were to apply evolutionary search algorithms (i.e. genetic algorithms and genetic programming) to the approximation of a function for components' loads within helicopters from easily measurable parameters. Adopting this approach requires both the load components and the corresponding flight data (e.g. positions of controls, aircraft accelerations, etc.) to approximate a function. Once it was approximated then the function can be used to predict load components without the use of strain gauges and/or slip ring load measuring equipment.

6.4. FLIGHT PARAMETER DATA

The total helicopter data, which contains a wide range of helicopter manoeuvres, is composed of 48 channels of flight parameter data and three channels of helicopter load spectra measured by means of strain gauges. This part of the study attempted to develop a function approximation by means of a GA and/or GP for the strain on *the main rotor spider arm control.* This seemed to be the most difficult to predict in the previous attempts using other techniques (e.g., using artificial neural networks (ANNs)). Besides, twenty-one of flight parameter data out of forty-eight were used as input parameters since experts in this field suggested that the rest of the parameters have almost no

effect on strain in the airframe or rotorhead (*Lison 1994*). These parameters and their sample rates are shown in Table 6.1.

It was aimed to reproduce a small part of the strain signal (i.e., the first 300 points) not only to get an insight into the characteristics of the problem but also to compare performances of the GAs and/or GPs with the other method, (i.e., ANNs) used in the previous work. It was also suggested that an 'artificial' parameter, namely, *sawtooth waveform at rotor frequency* (v_0), would increase performance of the system if it is used as an input parameter. 'Outside air temperature' was replaced with this 'artificial parameter' since it had a virtually constant value in the range used. As can be seen from Table 3.1, the data was sampled in two different rates and needed to be of the same size before being handled by the applications developed. The low rate channels were interpolated since the strain to be predicted was sampled in the high rate channels and preserving as much information as possible may be vital for accurate prediction. Furthermore, all channels were scaled between 0 and 1 because it is best to use the same order of magnitude for all parameters. They were scaled as follows :

$$x_{new} = \frac{x_{old} - x_{min}}{x_{max} - x_{min}}$$

Number	Description	Rate (Hz)
Input 1 (v_1)	Collective Lever Position	26.25
Input 2 (v_2)	Collective Servo Position	840
Input 3 (v_3)	F/A Cyclic Stick Position	26.25
Input 4 (v_4)	Control Cyclic Stick Position	26.25
Input 5 (v_5)	F/A Cyclic Servo Position	840
Input 6 (v_6)	Control Cyclic Servo Position	840
Input 7 (v_7)	Tail Rotor Pitch Angle	26.25
Input 8 (v_8)	Rudder Pedal Position	26.25
Input 9 (v_9)	Barometric Altitude	26.25
Input 10 (v_{10})	Barometric Airspeed	26.25
Input 11 (v_{11})	Lateral Acceleration	26.25
Input 12 (v_{12})	Main Rotor Speed	26.25
Input 13 (v_{13})	Outside Air Temperature	26.25
Input 14 (v_{14})	Pitch Attitude	26.25
Input 15 (v_{15})	Pitch Rate	26.25
Input 16 (v_{16})	Roll Attitude	26.25
Input 17 (v_{17})	Roll Rate	26.25
Input 18 (v_{18})	Aerodynamic Yaw	26.25
Input 19 (v_{19})	Normal Acceleration Centre of Gravity	26.25
Input 20 (v_{20})	Yaw Rate	26.25
Input 21 (v_{21})	Main Rotor Hub Torque	840
Output	Main Rotor Spider Arm Control	840

Table 6.1. Flight Parameter Data

6.5. EXPERIMENTAL DESIGN

Because the data are noisy, the fitness function is selected as the absolute correlation coefficient between the output of the models and the desired data.

Firstly the GP-1 and the GP-2 are used to approximate a function in symbolic form for the desired data.

The following designs were used:

- The GP-1

Population size	660 (Max.)
Max. generation number	3000
Length of an individual	640 (Max.)
Fitness required for the best	0.950
Crossover probability	0.750
Mutation probability	0.003
Desired data	Main Rotor Spider Arm Control
Number of parameters	21 (Defined in 6.4) v_0,,v_1--v_{12}, v_{14}-v_{21}
Amount of data in a set	300
Basic functions	+,-,*,/,sin(x,y,z),sin(x),tan(x), sqr(x), and sqrt(|x|). sin(x,y,z)=x*sin(y*2*π+z)
Initial population	Random
Decoding constants	as is in chapter 4
Fitness function	Absolute correlation coefficient

- The GP-2

Population size	100 (Max.)
Max. generation number	3000
Max. level of initial tree	3
Max. level of a tree generated for mutation	2
Crossover probability	0.750
Mutation probability	0.003
Fitness required for the best	0.950
Desired data	Main Rotor Spider Arm Control
Number of parameters	21 (Defined in 6.4) v_0,v_1--v_{12}, v_{14}-v_{21}
Amount of data in a set	300
Basic functions	+,-,*,/,sin(x,y), sin(x), tan(x),sqr(x), and sqrt($\lvert x \rvert$). sin(x,y)=sin(x*2*π+y)
Initial population	Random
Decoding constants	as is in chapter 4
Fitness function	Absolute correlation coefficient

6.6. RESULTS

- The GP-1

The best fitness in the initial population was 0.143.

The best function that the GP-1 found is:

$f(v_0,v_1,v_2,v_3)=$

$((\text{sqrt}(\text{sqrt}((\tan(\text{sqrt}(1.720))-1.770)))+\sin(v_2,((v_1-v_0)/v_3*\text{sqrt}(v_2)),v_2)*$

$(0.260-\sin(0.280,s((1.690/v_3),v_3,(1.690/v_3)),0.280)))-1.110)$

The maximum fitness: 0.766

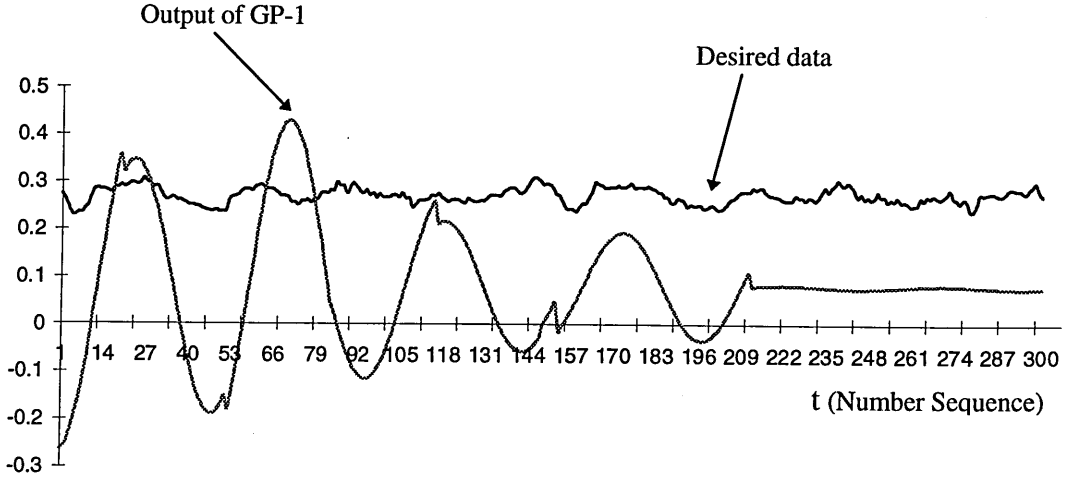


Figure 6.1. The desired data and the best output of GP-1.

- The GP-2

The best fitness in the initial population was 0.554.

The best function that the GP-2 found is:

$f(v_0,v_2)=$

$\sin((\sin(v_0*2*\pi+\text{sqr}(\text{sqr}(\text{sqr}(\text{sqr}(\text{sqr}(\text{sqr}(v_2)))*(\text{sqr}(\text{sqr}(\text{sqrt}(\text{abs}(v_2))))*$

$\text{sqr}(\text{sqr}(\text{sqr}(\text{sqr}(v_0))))))))*2*\pi+\text{sqr}(\text{sqr}(\text{sqrt}(\text{abs}(v_2)))))$

The maximum fitness: 0.716

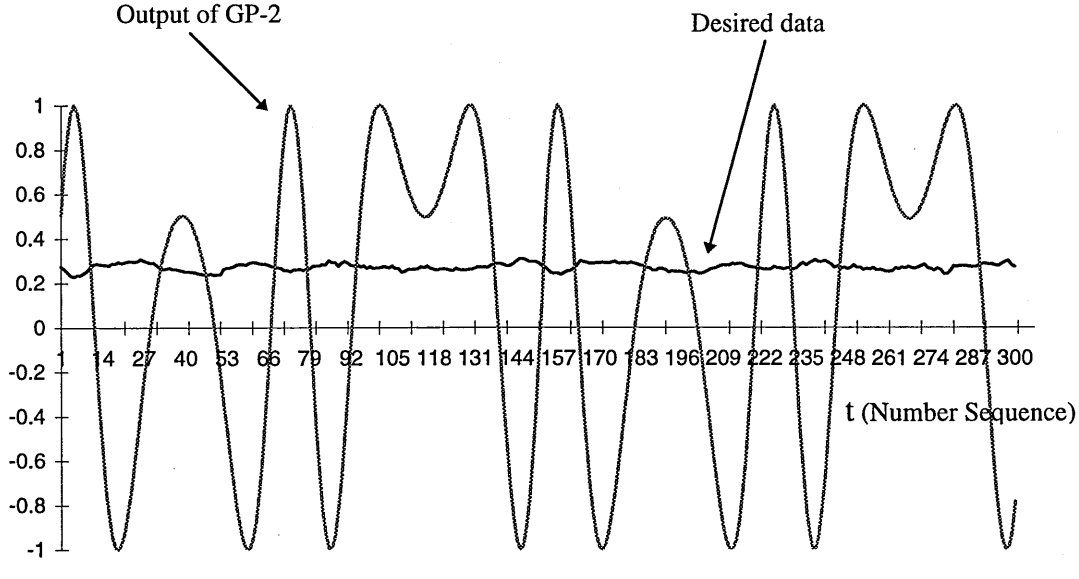


Figure 6.2. The desired data and the best output of GP-2.

CHAPTER 7

CONCLUSIONS

7.1. INTRODUCTION

This chapter discusses the results of the experiments which were undertaken to investigate Genetic Algorithms (GAs) and Genetic Programming (GP) in order to perform symbolic regression, and the overall performance of the system of approximating a function for the helicopter strain so as to draw a conclusion from the dissertation. First, the different models which were designed to implement genetic algorithms and genetic programming are revisited and evaluated. Secondly, potential further developments of the work are presented. Finally, the results from the study are assessed to conclude how well the aims are met.

7.2. DISCUSSION on the MODELS

In this research where GAs optimise the coefficients of predicted functions and GP approximates a function in symbolic form for data sets in hand, two different types of GAs and GP were tested for symbolic regression. Each type was tried with different population sizes, different crossover and mutation probabilities, and different ways of calculating fitness. The data sets (i.e. parameters of functions and function outputs) were generated using a computer program. In chapter 6, the data sets which were taken from a real world problem, helicopter health and usage monitoring, have been used to approximate a function for the strain on a helicopter's main rotor.

For the experiments, two different types of GAs and GP were designed. Although the types of GAs only differ on implementing the crossover operator, the types of GP differ on implementing the structure of the individuals and applying crossover and mutation operators.

In the first model of GAs (The GAs-1), the crossover operator of GAs is applied to an individual string at any point on a bit-by-bit basis.

The second model of GAs (The GAs-2) differs from the first model on implementation of the crossover operator. One of the boundaries between the parameters represented in the individual is selected as the crossover point. The crossover is not applied at any point within the parameters which are represented in an individual string.

Both models use a fixed length bit string representation.

The experiments show that the GAs-1 and the GAs-2 are working. They can be applied to problems of optimising coefficients of a function. The experiments also show that The GAs-1 improves the output of the model according to a fitness function better than the GAs-2. It is found that a strategy that starts by using correlation coefficient and then a fitness that combines both correlation coefficient and error worked better.

Instead of using the tree representation which is defined in the literature for GP, a fixed length binary string representation was used in the model GP-1. The aim was to apply the crossover and the mutation methods which are defined for GAs to GP. So that, GP can be tested using a different structure. The model tries to get a meaning from a string according to the *Polish Notation.*

The experiments show that the model can approximate a function in symbolic form to a desired data set if the fitness function drives the model well (e.g. absolute correlation coefficient). It is found that if the expected function is a complex one, the length of an individual string must be as big as possible.

The model GP-2 uses a tree to represent an individual of a population. The experiments show that if the fitness function is absolute correlation coefficient, the model finds a good solution. If the fitness is error-driven, the model cannot work well.

In the model GP-2, if the desired data set is a complex one, the trees always grow. This increases the memory requirement for the genetic process as the most complex ones have more chance to be copied to next generation. The model GP-1 has no problem of that kind as it uses fixed length representation. But if the maximum length of an individual is not enough to represent the desired data, the model is not going to find the expected function.

7.3. DISCUSSION on the HELICOPTER EXPERIMENT

The GP-1 and the GP-2 were used to approximate a function for a helicopter's 'Main Rotor Spider Arm Control'. The best result was obtained from the GP-1. The correlation coefficient was 0.756. Because the correlation coefficient was so low, the GAs-1 and the GAs-2 have not been used to optimise the coefficients of the functions which were found by the GP-1 and the GP-2.

The previous work (Karakoy 1996) which has been focused on the predicting of 'Main Rotor Spider Arm Control' using Artificial Neural Network shows that it can get better results in terms of the correlation coefficient and the mean absolute error for first 300 data. The correlation coefficient was 0.95.

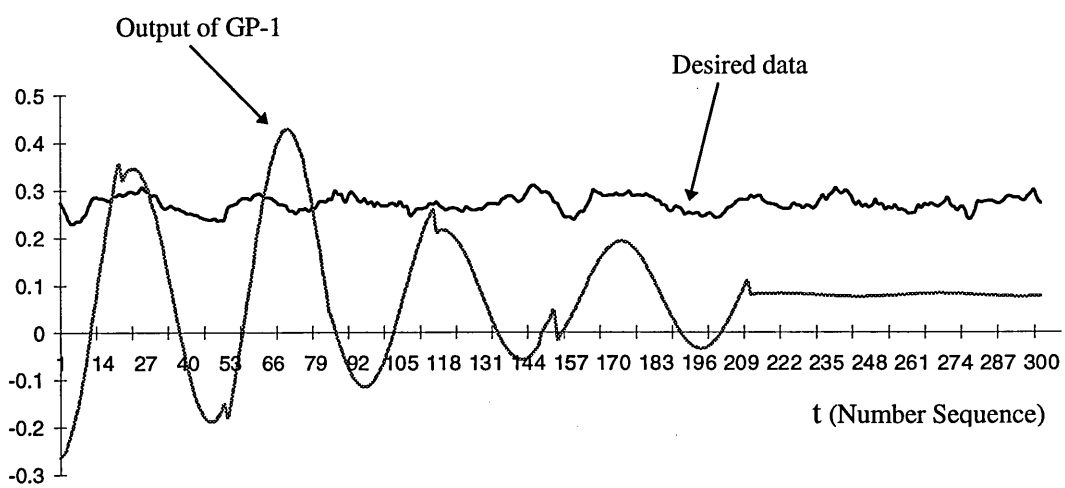


Figure 7.1. The desired data and the best output of GP-1.

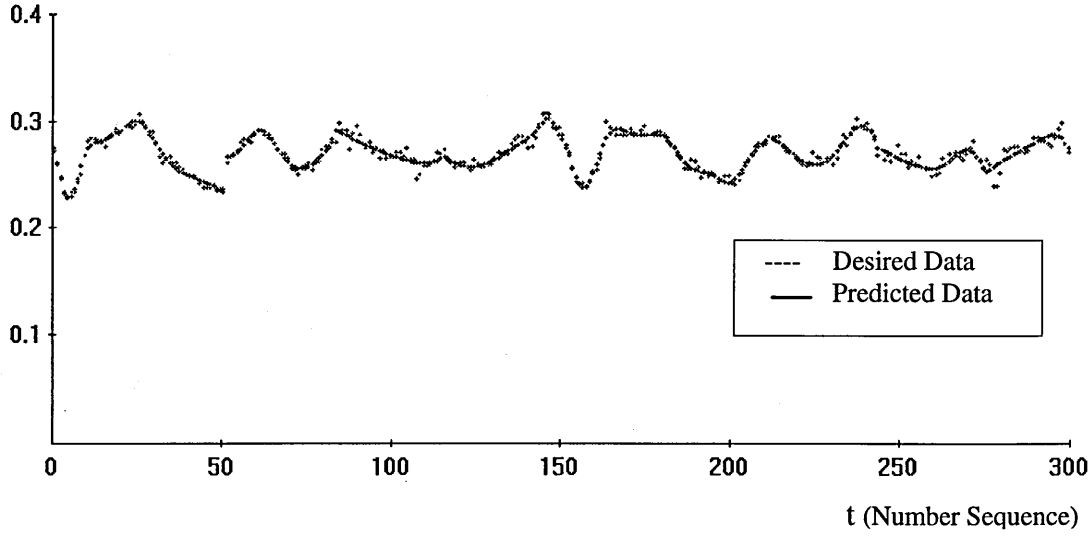


Figure 7.2. Training Result after 210,000 Iterations using Artificial Neural Networks (Multi-Layer Backpropagation '*MBPN*').

7.4. FURTHER WORK

The models which were designed to investigate genetic algorithms and genetic programming may be added new features in order to perform symbolic regression well. New strategies to drive the models to the best solution must be invented (i.e. fitness).

Using parallel algorithms with a new representation to implement GAs and GP may get better results. This must be tried.

Using GAs and GP with other AI methods and forming a hybrid system may lead to a better model. For example, after getting a function with a better correlation coefficient using GAs and GP, artificial neural networks may be trained using this function.

7.5. CONCLUSION

This work showed that GAs and GP can perform symbolic regression. In order to get better results, the right genetic parameters must be found. In order to drive the models to the best, fitness function plays an important role.

Although the GP-1 and the GP-2 could not find a good solution to the helicopter problem compared to the neural network's one, if the models are improved by adding new basic functions, using more memory in the search, they may work better than neural networks.

It is the author's belief that further development of the work will increase the models' ability to perform symbolic regression.

References

Baron, D.W. 1968 — Reqursive Techniques In Programming. Macdonald & Co. (Publishers) Ltd.

Bateman, R.D. 1982 — An Investigation into the Problems of Helicopter Maintenance and Maintenance Programmes, MSc Thesis, College of Aeronautics, Cranfield Institute of Technology.

Bayer, S.E.; Wang, L. 1991 — Problem Solving with Genetic Algorithms and Splicer. By AIAA-91-3836-CP.

Crawford, C.C., Carlson, R.L., Bates, P.R. 1991 — Damage Tolerance Analysis for Rotorcraft - What the Issues are, European Rotorcraft 17th Forum, Berlin, Germany, September.

CAA 1993 — Civil Aviation Authority. Usage Monitoring Working Group Report to the HHMAG, presented at CAA Aviation House, Gatwick Airport. Nov.

De Jong, K.A. 1975 — An Analysis of the Behaviour of Genetic Adaptive Systems. Dissertation Abstracts International 41(9), 3503B.

Deshpande, S.M.; Kumar, R.R.; Seywald, H. 1992 — Air Data System Optimisation Using Genetic Algorithm. By AIAA-92-4466-CP.

Goldberg, D.E. 1989 — Genetic Algorithms in Search, Optimisation, and Machine Learning. By Addison-Wesley Publishing Company, Inc.

Goldberg, D.E. 1994 — Genetic and Evolutionary Algprithms Come of Age. Communications of the Assocation for Computing Machinary. Vol. 37. P 113-119.

Hays, W.L. 1988 — Statistics. Holt, Rinehart and Winston Inc. Fourth Ed.

Hill, K.W.M. 1994 — The Application of Usage Monitoring to Military Helicopters. Cranfield Uni. Msc. Thesis. Aug.

Hill, K.W.M., Hudson, B., Irving, P., Vella, A.D. 1995 — Loading Spectra, Usage Monitoring and Prediction of Fatigue Damage in Helicopters, Proceedings of the 18th Symposium of the International Committee on Aeronautical Fatigue, Vol.1, 3-5 May.

Janson, D.J.; Frenzel, J.F. 1993 — Training Product Neural Networks with Genetic Algorithms. IEEE Expert, October 1993. P27.

Karakoy M., 1996 — Modelling Strains in Helicopters Using Artificial Neural Networks. MPhil Thesis. Cranfield Uni.

Kenneth K.E.Jr., 1993 Proceedings of the Fifth International Conference on Genetic Algorithms. Generality and Difficulty in Genetic Programming: Evolving a Sort. Morgan Koufmann Publishers.

Koza, John R. 1992 Genetic programming : on the programming of computers by means of natural selection. Cambridge, MA : MIT Press.

Koza, John R. 1994 Genetic Programming II - Automatic Discovery of Reusable Programs. Cambridge, MA : MIT Press.

KrishnaKumar, K. 1992 AIAA guidance, navigation and control conference, 1992, Hilton Head Island, South Carolina.

Lison, A.C. 1994 Flight Parameter Data Using Artificial Neural Networks. Msc Thesis. Cranfield University. Sep.

MOD-PEDS 1989 Ministry of Defence Procurement Executive (UK) Defence Standard 00-970 V.2 Leaflet 727 Series 'Health and Usage Monitoring Systems'. Sep.

Pham, D.T.; Liu X. 1995 Neural Networks for Identification, Prediction and Control. Whitaker UK.

Siegel,A.F.; Morgan,C.J. 1996 Statistics and Data Analysis. John Wiley&Son Inc. Second Ed.

Smith, G.D. 1994 Commercial Applications of Genetic Algorithms, University of East Anglia.

Srinivas, M.;
Patnaik, L.M. 1994 Genetic Algorithms:A Survey. IEEE Computer. June 1994.

Stockton, D.J.;
Quinn L. 1993 Identifying Economic Order Quantities Using Genetic Algorithms. International Journal of Operations & Production Management, V.13, N.11, p.93-103.

Vella, A.D.;
Hudson, R.A.;
Irwing, P.E. 1996 Estimating Helicopter Strain Using Neural Network Approach, in 'Neural Networks and Their Applications'. Edi. Taylor, J.G.&Willey.

Wendlandt, N. 1990 Engine Condition Monitoring Applied to Life Usage of Military Aero Engines, MSc Thesis, SME, Cranfield Institute of Technology, 1990

Willian, B.L.;
Qureshi, A. 1996 Computers Using "Natural Selection" to Generate Programs. Dept of Computer Science, University College London.