ISSN 0280-5316
ISRN LUTFD2/TFRT--5642--SE

# Genetic Programming in Control Theory: On Evolving Programs and Solutions to Control Problems

Kuan Luen Ng

Department of Automatic Control
Lund Institute of Technology
June 2000

|  |  |
|---|---|
| **Department of Automatic Control Lund Institute of Technology Box 118 SE-221 00 Lund Sweden** | *Document name* MASTER THESIS |
|  | *Date of issue* June 2000 |
|  | *Document Number* ISRN LUTFD2/TFRT--5642--SE |
| *Author(s)* Kuan Luen Ng | *Supervisor* Rolf Johansson, LTH |
|  | *Sponsoring organization* |

*Title and subtitle*

Genetic Programming in Control Theory: On Evolving Programs and Solutions to Control Problems. (Genetisk programmering i reglerteori).

*Abstract*

More often than not one would encounter a problem, know that the solution has to meet some requirements, but do not how to start or how to progress towards solving it. Motivation for a computer that can solve the problem automatically without explicitly programming it is apparent, i.e. a computer that \programs itself", is greatly desired.

The method of genetic programming has demonstrated its potential by evolving programs for a wide range of applications. Examples are target identi_cation [Tackett, 1993], performing optical character recognition [Andre, 1994], electronic circuit design [Koza, 1996] among many. In certain areas, GP generated designs or solutions were shown to be on par or even better than those created by human; although this is of course, not always the case. However, the power of GP is inherent that it is possible to use all tools and functions of computer programming that have ever been devised since it is evolving programs from programs themselves.

This thesis applies this method to the area of control engineering. Applying genetic algorithm to this _eld is not new, however, but using genetic programming is relatively recent. The intention is therefore clear, to introduce another set of tools, perhaps quite unconventional but hopefully useful, to the control engineer.

All the solutions presented in this thesis have been implemented using the program Matlab. Although the mathematical functions that have been used are limited and often no more than simple additions or multiplication, it should be clear that a whole arsenal of Matlab functions could be used as part of GP, depending on how the programmer formulate his/her problem. Hence besides having the basic program code to be re-usable for di_erent problems, care has been taken to allow user to be able to add additional functions easily. Designing a GP run would then consist of simply selecting or writing functions, and setting up a suitable evaluation and termination criteria. Problem solving should be automatic. Perhaps one might think that the solutions presented here are trivial, and the program might not work for other more diÆcult problems, or it may end up in a combinational explosion. This should however not to be seen as a poor re ection on the method, but rather may simply be due to bad programming practice. Therefore having said that, it should also be mentioned that there is more than enough room for improvements to the basic program

*Keywords*

*Classification system and/or index terms (if any)*

*Supplementary bibliographical information*

|  |  |  |
|---|---|---|
| *ISSN and key title* 0280-5316 |  | *ISBN* |
| *Language* English | *Number of pages* 101 | *Recipient's notes* |
| *Security classification* |  |  |

*The report may be ordered from the Department of Automatic Control or borrowed through:*
*University Library 2, Box 3, SE-221 00 Lund, Sweden*
*Fax +46 46 222 44 22 E-mail ub2@ub2.se*

# Genetic Programming in Control Theory:

On Evolving Programs and Solutions to Control Problems

Department of Automatic Control,
Lund Institute of Technology
&
Electrical and Electronic Department
Imperial College of Science, Technology and Medicine

Kuan Luen Ng

June 20, 2000

# Acknowledgement

This thesis is a one year long project for ERASMUS exchange programme from my home university, Imperial College, London, to my host university, Lund Institute of Technology, Sweden.

I would like to thank my supervisor, Rolf Johansson, LTH, for his kind guidance and care; Philippe De Wilde, IC, coordinator for the exchange, and Kalle Åstrom, LTH, lecturer on Optimisation course, for their ideas and critisims; Per Hangander, director of studies, for his troubles and patience on the coordination side of the exchange and thesis.

My thanks to Stepfan Solyom, Mattias Grundelius and Bo Lincoln, PhD students, for their help in various topics such as Adaptive Control and Nonlinear Control. Leif Andersson, for his help on computer systems, Elektroteknik, Teknisk fysik och Datateknik and Department of Automatic Control for use of computers.

Thanks also to Henrik Olsson and Xu XiaoDong for reading and constructively criticising my drafts of the thesis.

# Contents

# Chapter 1

# Introduction

Genetic Programming [Koza, 1992], GP in short, is a method using Darwinian idea of natural selection to create a computer program automatically from high-level statements of problems' requirements. GP itself is an extension of Genetic Algorithm (GA), first developed by John Holland (1975), who mimicked the way bio-organism evolved according to the rule of 'the fittest survives'. The way nature has it that a particular species flourishes, is by inheriting the genes from the fitter individuals, from one generation to the next. In the case of GP, these individuals are sets of programs that are being 'cross-breed'.

Typical runs of a genetic programming system include first the preparatory steps of defining the problem statement communicated through a 'fitness function', functions that the system can use and other design parameters. During the execution, individuals are selected to participate in various 'genetic operations' and new generations of individuals are produced in this manner. When certain terminal criteria, such as when the solution has been found, is met, the run will stop and the results is one that the individual gives the satisfactory output. Genetic programming is a competitive beam search among a diverse population directed to the goal of discovering a satisfactory program that will solve the given problem.

# Part I

# Part I: Background

# Chapter 2

# Genetic Algorithm

Genetic Algorithm (GA) derives its name from the pioneer John Holland [Holland, 1973] who got his inspiration from the way of natural evolution. The method starts with a population of objects, or individuals, each having a known fitness. Using Darwinian principle of survival and reproduction of the fittest, genetic operations such as crossover (sexual recombination) and mutation are performed on these individuals, and a new generation of individuals are bred from the pervious generation. Using a suitable fitness function that gives a probability for which individuals to choose from, Genetic Algorithm solves a problem through driving the population's evolution in the direction we want it to go.

In application, very much like DNA in nature, GA uses an encoding scheme to represents a probable solution, typically a vector or simply just binary code. In optimisation and other problems, this can also be a point in a search space. Individuals with these 'DNA' records could therefore be a used as criteria of judging the individuals' fitness or how close they are to the solution. [Langdon, 1998]

The mechanics of a simple genetic algorithm is surprisingly simple, and an example is shown below. As would be seen, the method involves nothing more complex than copying strings and swapping partial strings. A suggestion on why this simple process works is more subtle and powerful, this would be presented in later sections. However it should be noted that the attractiveness of this method lies in its simplicity of operation and effectiveness. [Goldberg,1986]

## 2.1 Simple Genetic Algorithm: example

As mentioned the breeding of a new generation is inspired by nature. Taking say a set of strings of binary code, new strings are bred from the fitter ones in the current generation. This can be done using either asexual or sexual reproduction. In asexual reproduction, the parent string is simply copied into the new generation. Mutation, such as a random change in the binary number, can also occur. Arguably the role of mutation provides certain randomness that is present in nature, and its effect is seen in some cases where convergence to solutions is faster than not having it. In sexual reproduction, two of the fitter strings are chosen and the new string is created by sequentially copying bits of binary code alternatively from each parent. Typically two or three parents are used, and the point(s) of crossover on the binary string is chosen at random. see fig 2.1 Figure shows an example of how this is done. A newborn is created by firstly copying 3 binary numbers (or ‘genes’) from the 1st parent and 4 binary numbers from the 2nd parent.

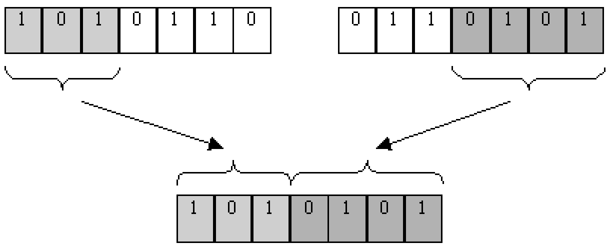


Figure 2.1: Genetic Algorithm - crossover operation

Holland [1973] shows via his schemata theorem that in certain circumstances genetic algorithms makes good use of information from the search at that point to guide the choice of its new points to search. Goldberg [1987] gives a more intuitive approach to the theorem, which will be described later in section 7.1.6. It should be able to give an insight to why genetic algorithm works and also show the power of such search techniques. However it is not to say that this method is universally superior. Nonetheless, firstly we can look into the central theme of genetic algorithm and see what it strives to be good at.

## 2.2 Robustness

Genetic algorithm tries to surpass its cousins of search techniques in the area of robustness. The balance between efficiency and efficacy necessary for survival in many different environments takes central stage. Implications of robustness for artificial systems are of course self-apparent. Costly re-designs can be saved or eliminated, higher level of adaptation can be achieved, and existing systems can perform better and longer. In control theory, imagine if this method could be applied to the area of automatic control, where genetic algorithm could provide the additional supplementary features of self-repair, self-guidance, better robustness and reproduction which is present in biological systems. This is not pursued in this thesis but its implication should not be ignored. In short, nature does it better when it comes to robustness.

It would not make sense either, if we just accept the method of genetic algorithm simply because it mimics the beauty of nature. As Holland and Goldberg had shown, genetic algorithm is theoretically and empirically proven to provide robust search of complex spaces. Also, the method is not fundamentally limited by restrictive assumptions about the search space, such as those concerning continuity or existence of derivatives. The real world of search is such, very often full of discontinuities, noise and multi-modal. This of course, does not mean that other methods are not good; it simply means that some are better in solving certain problems over the others, while they might not perform just as well in other sets of problems. For this the next section provides a brief description of different search methods available and then the 'No Free Lunch' theorem will be mentioned. This should provide some basic idea what we should and should not expect from genetic algorithm and genetic programming as well.

## 2.3 Search Techniques

This thesis is not a comparative study of search and optimisation techniques. However it is necessary to mention them in order to appreciate that certain methods perform well in some problems while perhaps not in other areas. Since robustness is the main concern for genetic algorithm techniques, some focus would be given to that. However in short it should provide some insight to both the power and limitations to our search techniques.

Currently there establish three main categories of search methods: calculus based, enumerative, and stochastic. Further denominations can be separated from these basic three. The categories can be shown in the fig 2.2.

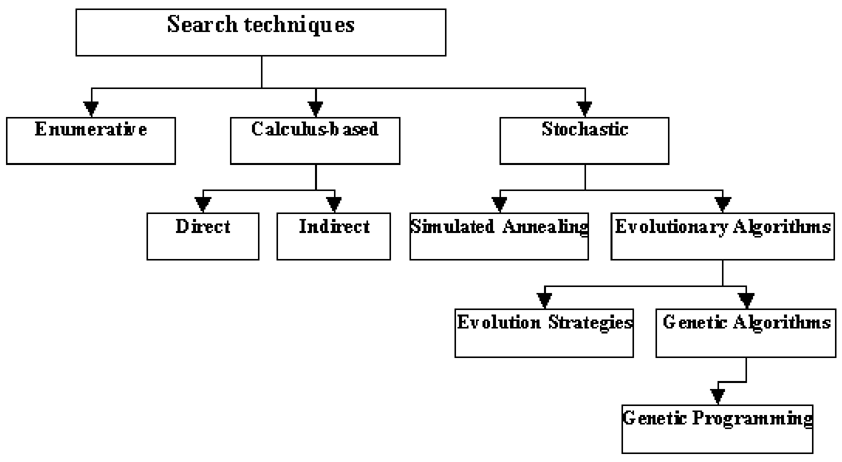


Figure 2.2: Search Techniques

**Enumerative** method involves searching all possible points one point at a time within the search space. The problem in such techniques of course lies in its lack of efficiency, the practical search space might be too large and there might not be any chance where the search can use some information to itself along the way. This technique however, is easy to use.

**Calculus-based** techniques are well known. In general there are two main classes, namely, direct and indirect methods. Both techniques treat the search space as a continuous multi-dimensional function and look for the maxima (or minima) using its derivatives. In Indirect search, the idea is to seek the local optima by solving usually a set of equations to find points where derivatives would be zeros. The function need to be smooth and for a large area of search space, number of samples or points where derivatives are zeros could be small. In direct search, common techniques such as Fibonacci and Newton methods seek local optima by hopping on the function and moving in the direction of the local gradient. These techniques are known as hill-climbing methods because from evaluating the current point, they tend to climb the function in the steepest permissible direction to the subsequent point. Calculus based techniques have improved over the years, and some complex problems can be transformed into 'better-behaved' problems where the methods can then be applied. Nonetheless when the search space is filled up with undesirables such as noise and discontinuities, these techniques might become insufficiently robust in the domain.

**Stochastic** techniques use information from the search so far to guide the probabilistic choice of the next point(s) to try. Such search recognise the shortcomings of calculus-based and enumerative schemes, and are more general in their scope. Simulated annealing uses random processes to search for minimum energy states using the physical annealing process [Davis, 1987]. Evolutionary algorithms based their method in Darwinian theory of evolution where the fittest survive. In evolutionary strategies, typically the search space consists of vectors of real values. Adding random noise to the current points on these vectors, new points are created. The search will continue from the new points if they are better than the old, if not the old ones are retained. Finally, in genetic algorithm, the search space is characterised by usually a vectors of bit codes. New vectors, which may potentially lead to the solution, are created from the bit codes of the parent. This is analogous to the way chromosomes of DNA are passed to new generations from the parents.

## 2.4 Expectation v.s. Non-Free Lunch

Goldberg [1987] suggested that genetic algorithm could surpass their traditional cousins in the area of robustness. Calculus-based methods perform well in a narrow problem class, as would be expected, but become inefficient in other bands of problem. Enumerative techniques or simply a random walk would be almost equally inefficient across the whole band. The ideal desire is for which genetic algorithm would hopefully fulfil, is a technique that has a relatively high performance across the whole spectrum of problems. It would then be worthwhile to sacrifice peak performance on particular problems for generality.

We now know that although any of the above search methods may be well-suited to a particular problem, the No Free Lunch (NFL) theorems [ Wolpert and Macready, 1995] showed that averaged over all possible problems the techniques are all equivalent. That is if one performs highly in one particular problem, there exist other problems where it will perform badly. (Of course these other problems might not be of any original interest). Its implication could be then that when averaged over all possible problems, any search techniques, including genetic algorithms (and of course genetic programming too), would perform just as badly as random search. This is of course heartbreaking to know but nonetheless NFL theorems provide a formal way to argue that the search techniques presented in this thesis should match the search problems, perhaps sometimes perform better whereas sometimes not, compared to other techniques, and show that there is no universally good algorithm for all possible spectrum of problems.

Furthermore, while genetic algorithms have found their application in a wide area of research, genetic programming (although the difference might be minute) is relatively new, and as far as we know, not very common in the application to control engineering. As such, most of the work presented in the thesis is exploratory. Since it is so, there is no guarantee of results that may even be close to satisfactory. In such cases, the thesis will try to provide some explanation and discussion on why it did not perform as expected, or in the case of reasonable success, why it should work.

# Chapter 3

# Genetic Programming

## 3.1 History and Background

The idea of combining genetic algorithms and computer programs is not new, since the early days in the development of genetic algorithms. Different attempts were made but it was John Koza [Koza, 1992] who successfully applied genetic algorithms to program language LISP and showed that this form of methods can be applied to a wide range of problems. From then, number of papers on the subject has grown at exponential rates.

Applications using this method are wide. Although there may be some reservation on the technique's scalability to solving more complex problems, in terms of computational effort, genetic programming has been applied to a wide field of areas. Koza's latest book, Genetic Programming III: Darwinian Invention and Problem Solving [1999] for example, presented solutions to problems from the area of system identification, time-optimal control, classification, synthesis of cellular automata rules, synthesis of minimal sorting networks, multi-agent programming, and synthesizing both the topology and sizing of analogue electrical circuits. Nonetheless, having said that genetic programming showed in favour in solving a variety of problems, it should also be mentioned that however capable, it has not been demonstrated to be capable of solving all problems of all types from all fields.

A computer program can be thought as a particular set of solutions (or a particular point within) to a possible search space consisting of all such programs. Hence computer programming can be said to be searching for such a suitable program that might work, within this space. Human programmers when working on the task would use their skills and experience to direct their search so as to find correct program. Many tools and representations are available, such as high-level languages or code generators, for the human programmer to use to make it easier to complete the goal.

When programming neural networks, calculus based search techniques such as back propagation are often used. Simulated Annealing, Evolutionary Strategy and Genetic Algorithms have also been used to program artificial neural networks. Calculus based methods usually transform the search space so that it is smooth and solvable. When the neural network has terminated with the search, it is said that the network has been trained. A program in this sense has been automatically created.

Genetic programming on the other hand uses stochastic search technique. Instead of transforming the search space, such form of searching involves looking at the original space itself. However because of its discrete nature and a vast number of possible programs, enumerative search is tedious and inefficient. However, as genetic algorithms have demonstrated some successes as stochastic search techniques in solving problems, genetic programming applies similar strategies. Of course genetic programming, for it to justify its name further, also combine traditional programming methods such as code reuse, iterations, function definitions among many, in the quest for its search. In brief, we could say that genetic programming have achieved its success by combining genetic algorithms with traditional programming.

We now attempt to go into some detail proper on how programs can be 'breed', and some suggestions on how it can be applied to control engineering will be in the next chapter.

## 3.2 A Brief Introduction

Genetic programming is a technique that allows computers to evolve problem-solving capabilities without explicitly being programmed. It uses the method of genetic algorithm to automatically search and generate the required suitable program. Genetic algorithms have been discussed in the previous sections; in the case of genetic programming, the individuals are now computer programs. A run of genetic programming is a competitive beam search among a diverse population directed to the goal of discovering a satisfactory program that will solve the given problem. The flow sequence of genetic programming is shown in Fig 3.1. For more details, please refer to [Koza, 1999, Langdon, 1998] and the bibliography.

## 3.3 Preparatory Steps

There are five basic preliminary steps to solving a problem using genetic programming. This is a reflection on what typically characterise machine learning. It could start with a human user asking something like, "How can computers be made to do what needs to be done, without being told exactly how to do it?" [Arthur Samueal, 1959]. A system that automatically creates computing programs and solutions should at least be told "what is it to do". Hence these steps provides some methods for communicating to the system before it start, what is the requirements of its action. (As a comment, if a system knows automatically what is it to do, it should denotes then, some form of consciousness. However artificial consciousness (see Alexander. 1997) is not the focus of this thesis and perhaps genetic programming in general now.

These five steps, directly from Koza's terminology, are:

1. set of terminals (e.g. the actual variables of the problem, zero-argument functions, and random constants, if any) for each branch of the to-be-evolved computer program.

2. the set of primitive functions for each to-be-evolved branch

3. the fitness measure (or other arrangement for explicitly or implicitly measuring fitness),
4. the parameters for controlling the run, and
5. the termination criterion and the method of results designation for the run.

Originally, there is the sixth step which involves determining the programs' architecture (6). In Koza's work this means defining the number of automatically defined functions (ADFs), the number of arguments they take, which may call which one, and which may be called from the main program. More recent works [Koza and Andre, 1995] have shown that the architecture itself can be evolved during a GP run. Such choices of multi-tree program architectures are similar to that in this sixth step. As this thesis uses this step to some extent, it should therefore be mentioned.

### Terminals and Functions

The first two steps concern the ingredients that are to be used to create the computer programs. We have to decide the terminals and functions that the evolved programs will be composed of, so as to ensure that they are capable of expressing the solution to the problem. Often these may simply be the four arithmetic operations of addition, subtraction, multiplication and division, a conditional branching operator, the inputs and constants. These terminal and functions form an executable program that represents a trial solution. The transmission functions may consist of genetic operators (crossover, mutation etc), ADFs, fitness function and selection scheme. Through out the succession of generations, these transmission functions should drive the population towards an acceptable solution. Design of a successful set of these functions (let alone optimal) is not trivial. This could be seen from the subsequent experiment reports.

### Fitness Function

The third step involves designing the fitness criterion for investigating the trial programs. It is the high-level communication statements to the GP systems with regard to the requirements of the problems' solution. The first two steps involve defining the search space, while this third step affects the outcome of the search.

Fitness function is what drives the population towards the solution. Its purpose is not only to give high reward to trial solutions with the correct answers, but also reward reasonably solutions which has improved performance over the last generations. There are of course many ways which fitness functions can be proposed. However the main characteristic of the fitness function (and of genetic programming) is that the user specifies "what is to be done" but not reveal "how to do it". The fitness function does not give any hint on what to use for example, iteration or recursion, or what kind of approach it should take to get to the answer.

A fixed fitness function, or one which is variable as time passes, can be used in a GP run. Usually the fitness function carries out a trial of the evolved program. The individual is tested and its results could be judged against what is required. This gives a measure of how well the particular individual is.

A problem with such testing is that usually not all cases can be tested, simply because of the sheer amount of work required to do so. Even if a program is able to pass all the tests designed by the clever human programmer, there is always a risk that the program may contain error. Another problem is that since in most, if not all of the time, each individual undergoes the full set of tests. Since the volume of testing is sometimes already huge, with thousands or even millions of individuals in each generation the situation aggravates. Without doubt then, in most cases it should be expected that determining the fitness of trial solutions would take up the most amount of computation time in a GP run. Luckily, in this thesis most of the solutions could be reached with reasonable size of population and not too vigorous fitness functions.

It is possible to design sets of "fool" or inadequate fitness functions that will drive a population away from optimal solutions, towards some local optima. That is, the solutions may give relatively high fitness but do not solve the problem. These problems have been studied in the research of linear genetic algorithms [Goldberg, 1989, 1992 et al.]. Although it has been generally rejected as problem in genetic programming, it did occur in some cases of this thesis. However with some additional functionality and rearrangement of program structure, these difficulties became manageable. For now a brief summary of what was concluded of designing of fitness functions:

1. "Fool" or inadequate fitness functions can drive GP to sub-optimal solutions. The way around the problem can be to design better fitness functions or provide better (or more efficient) terminals and functions for the search space.

2. It may be easy for a trial solution to score well for some cases but not others. To do well in specific problems may make the solution lose its generality. A good fitness function should prevent sufficiently well such cases from occurring.

### Control Parameters

Many control parameters are involved in a GP run. Some are more important over the others. Population size however, is of first interest. Generally we would want a pool of possible solutions that is manageable in terms of computational time required for accessing it but yet gives a reasonable chance for diversity and reaching the correct answers. Other control parameters among others are the number of generations, depth and complexity of solutions, or percentage of the number of crossovers against reproduction.

### Termination Criterion

The most common termination criterion would be to stop the GP run when some particular individuals have reached the correct solution or pass the fitness function test. The GP run can be stopped too, when the total number of generations has been reached. The reason to do so is because in many problems, GP seems to give only marginally improved solution after certain number of generations. Koza [1992] argues that in many cases it is better to run a GP several times rather than increase the number of generations in one run. It was found to be generally true in this thesis.

## 3.4 Execution Steps

After the preparatory steps are completed, the run of genetic programming is ready to be launched. Execution steps are a series of actions that are performed in a loop and will terminate when the number of generations is reached or when the solution to the problem is reached. As like that of preparatory procedures, the execution steps of GP contain no discretionary human intervention or interactions during the run. The basic form is as follows:

1. Randomly create an initial population of individual programs
2. Iteratively perform the sub-steps below (in one generation) on the population till the termination criterion is satisfied.
   - execute each program in the population and assign a fitness value using the fitness measure prepared.
   - Select one or two individual program(s) from the population probabilistically based on fitness to participate in genetic operations described next. Reselection is allowed.
   - Create new individuals from the old using one of the genetic operations with which are also based on specific probability. The operations are:

     **Reproduction:** Copy the selected individual into the population.

     **Crossover:** Create new offspring program by combining randomly chosen parts from the two selected parents.

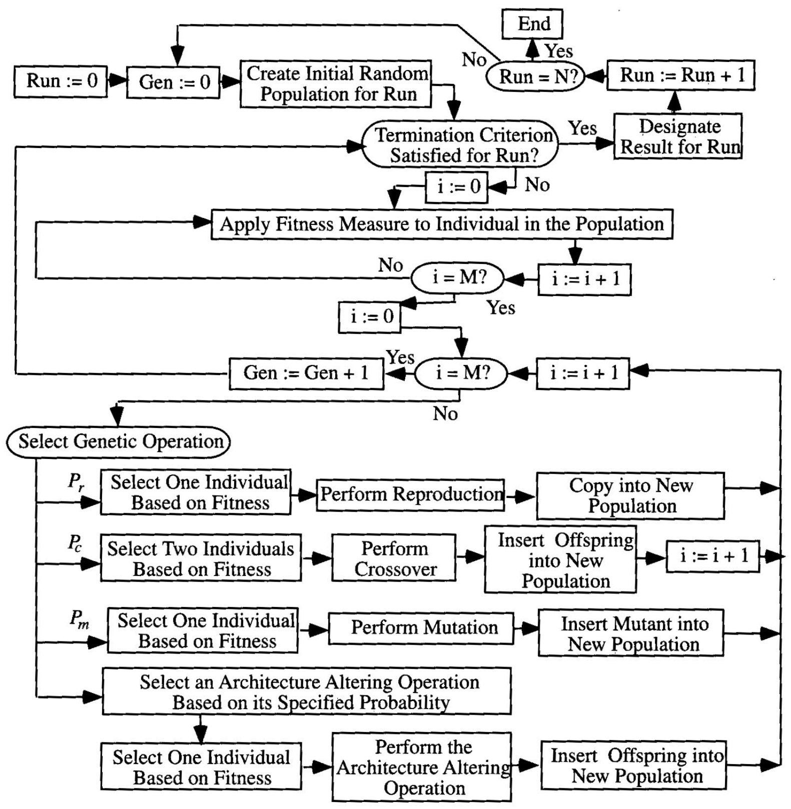
     **Mutation:** Create new offspring program by randomly mutating a randomly chosen part of one selected program.

     **Architecture-Altering Operation:** Create new offspring program by using on of such operation on one selected program.
3. After the termination criterion is met, the run is stopped. This may occur when the number of generations is met, or when the solution is met with one of the individual program. Usually the best individual (best-so-far) is returned as the result of the run. This may or may not be the satisfactory (or an approximate) solution to the problem.

Figure 3.1 shows the flow-chart of a GP run. It should be commented that it is similar to that of a basic genetic algorithm procedure, except for the case of GP, besides individuals are programs instead of vector strings, there exists the additional step of Architecture-Altering Operations.

## 3.5 Initial Generation

The initial population of the computer programs can be composed of a random pool of functions and terminals appropriated for the problem. There can exist a single or multiple outputs from the program. Also, the size of the initial



End
Yes
No
Run = N?
Run := Run + 1
Run := 0
Gen := 0
Create Initial Random Population for Run
Termination Criterion Satisfied for Run?
Yes
Designate Result for Run
i := 0
No
Apply Fitness Measure to Individual in the Population
No
i = M?
i := i + 1
Yes
i := 0
Yes
Gen := Gen + 1
i = M?
i := i + 1
No
Select Genetic Operation
$P_r$
Select One Individual Based on Fitness
Perform Reproduction
Copy into New Population
$P_c$
Select Two Individuals Based on Fitness
Perform Crossover
Insert Offspring into New Population
i := i + 1
$P_m$
Select One Individual Based on Fitness
Perform Mutation
Insert Mutant into New Population
Select an Architecture Altering Operation Based on its Specified Probability
Select One Individual Based on Fitness
Perform the Architecture Altering Operation
Insert Offspring into New Population

source: Koza et al, Genetic Programming III

Figure 3.1: flow chart of genetic programming

program can be random or pre-determined up to the programmer. There is theoretically no adverse results in choosing the wrong program size to solve a problem, although it would be shown later it does affect the speed of reaching the correct solution.

## 3.6 Genetic Operations

Inspired by the working of nature, genetic operations in GP are the mechanisms that allows population to move towards the designated goal. The three main operations are reproduction, crossover and mutation.

### 3.6.1 Reproduction

The Darwinian principle of selecting the fitter of the individual is used, and a copy of the program is inserted in the next generation of the population.

### 3.6.2 Crossover Operation

The crossover operation acts on two parental programs selected also based on fitness and creates one (or two) new offspring programs consisting of parts of each parent. An easy way to demonstrate the basic idea of crossover in genetic programming would be to use an example.

#### Simple Genetic Programming: an Example

A program, say mathematical equations, can be represented more easily with tree diagrams such as those in Fig. 3.2. A genetic operation can be performed for example by taking some branches from the parent programs and inserting them into a new offspring. A new program with a different tree-like structure is now created. This is similar to the case of crossover in genetic algorithms.

For example, say a solution can be arrived by calculating $y = x^2 + x$ without us explicitly telling the computer that it is so. Our population of possible programs may, among a diverse selections, consists of two programs, $y = x/(x(x - x^3) + x)$ and $y = x - x + x + x = 2x$ . Both are selected from the population because they produce results that are close to $y = x^2 + x$. (See Fig 3.2 for the structure. Their equivalent graphical form is in Fig.3.4 ) A fitness criterion can be for example the sum of residual error between the individual program and the correct answer. Now a branch from each parent is selected, say from the father program $x \times x$ could be selected and inserted to the mother program at the point where the branch $+x$ remains (while the rest may be discarded). A new program is thus created and may now yield a higher fitness. In this case the particular program actually solves the problem and the GP run is terminated, producing the required output. See Fig 3.3

A sharp reader may notice, for example, that the solutions from GP may not be optimal, that is it may contain redundancies such as $(x - x)$ branches that represents nothing. Also, there are certainly more than one ways where combination to form the correct answer may be possible. A simple solution such as the one mentioned above just need an occurrence where a branch $x^2$ and $+x$ could be selected and combined. Also, most of the times, all the programs or individuals in the search space has to be valid and executable or the evolved

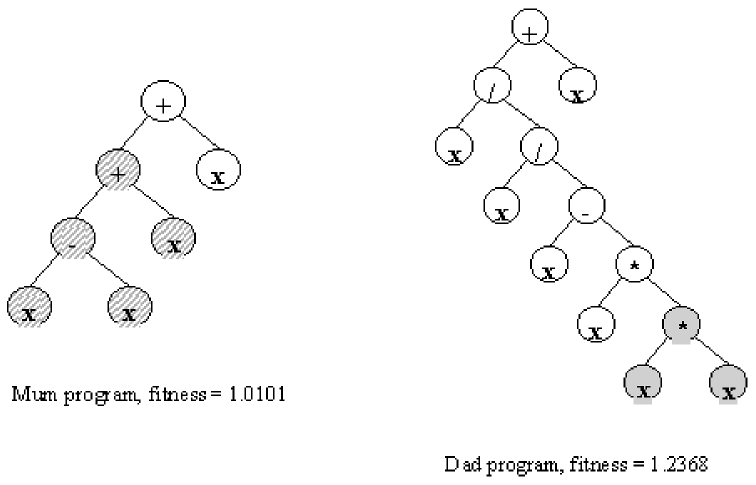


Figure 3.2: An example of tree-like structures in GP

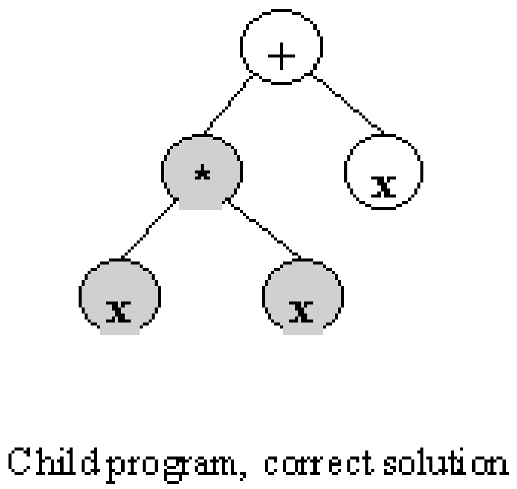


Figure 3.3: created offspring from the parent programs

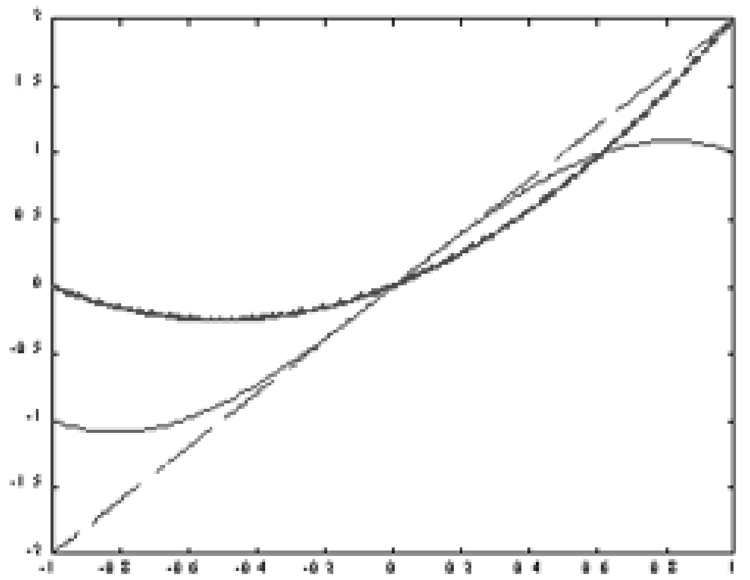


Figure 3.4: graphical representation of the family of programs

program will not work. This is true especially for example in cases where there might be divide-by-0 functions. This is of course, an inherent property that may exist in GP evolved programs, and problems need to be addressed. More would be discussed later with regard to closure and efficiency of programs

### 3.6.3 Mutation Operation

Mutation operation acts on one parental program selected based on fitness. As again inspired by nature, mutation means a random transformation within the genes of an individual. This operation played a role in other applications of evolutionary methods but not in the early works of genetic programming. Koza [1992, 1994] wished to show that GP was not performing a simple random search and hence mutation was omitted in GP runs. However, while mutation is not necessary for solving a wide variety of problems, mutation operations are now increasing being used in GP (including Koza). Many have claimed that mutation can be advantages instead of just doing simple crossover/ reproduction operations. [Chellapilla, 1997]. Mutation operations are used sparingly in this thesis, with similar purpose of showing that GP when applied to Control Engineering is also not a simple random search. However, since it may provide a further enhancement to the tools available, it is worth mentioning here. Some mutation operators are:

1. Sub-tree mutation: replaces a randomly selected tree/branch of the program within the individual with another.
2. Node replacement: or known as point mutation, replaces a particular node which is randomly picked within the program with another.
3. Mutation of Constants: mutates constants by adding Gaussian distributed noise to them.
4. Other operations such as Hoist, Shrink, Permutation.

## 3.7 Closure

Closure is defined by Koza [1992] as the property which each of the functions are able to accept as its arguments any value or data type that is possibly returned by any function including itself and given by the terminal. Closure is necessary so that when doing crossover, any arbitrary point or sub-tress can be combined and the resulting new offspring would still be correct and executable. Often closure can be achieved by requiring all terminals and functions to take and return the same argument types, e.g. integers. Special cases need to be considered such as divide-by-0 as mentioned before. In the cases of our experiments where this occurred, a divide-by-zero would return a 1. Functions such as these that take in illegal arguments and return valid results are called protected functions. Closure is an important aspect in genetic programming since it affects the choice of terminals and functions, hence presenting another difficulty in problem representation. This should be even more of a problem when more complicated programs are to be evolved.

## 3.8 Automatically Defined Functions

In computing, programmers often organise sequences of primitive steps into groups (subroutines, modules or procedures) for the purpose of code reuse and also breaking a problem into smaller sets of problems. In genetic programming, this is done using the mechanism of Automatically Defined Functions (ADFs).

An ADF consist of a function-defining branch that possesses zero, one or more variables (formal parameters from problems) and whose body is subject to evolutionary modification during the run of GP. Each of these ADFs resides in a separate function-defining branch within a multipart computer program. This means that it belongs to particular individual program within the population.

ADFs are used in some of the experiments in this thesis. However they belong to the first type introduced by Koza [1994], which is non-evolvable. However a structure for the evolvable type was laid down for easy enhancement in the future. This would be mentioned again later.

## 3.9 Architecture-Altering Operations

A system for automatically creating computer programs should require as little human user's help in pre-run decisions as possible. One of the shortcomings of existing techniques has been the requirement that the human user predetermine the size, shape, and character of the final solution to the problem. These factors ideally should be part of the answer produced by the automated system, and not part of the question addressed by the human user.

Architecture-Altering Operations (AAOs) provide the capability to automatically create ADFs, select size and architecture of the program. These operations are done concurrently in the evolutionary process of a GP run. In his latest work [Koza, 1999] described AAOs for subroutines such as ADFs, automatically-defined iterations, recursions and memory storage, together with examples of their usage. In brief, AAOs provide an automated way (from a high level point of view) to decompose a problem into a non-predetermined number of sub-problems and assemble them into a solution. They can also be viewed as a method to change the representation, generalisation or specialisation of a problem automatically.

During a GP run, in each generation a small number of the individuals are selected (on fitness) and have one AAO acts upon it. A crude form of AAO, namely subroutine duplication, had been tried out in our experiment with some slight successes.

## 3.10 Attributes of Genetic Programming

Given that one of the main purposes of genetic programming is to produce a computer program automatically, it should be mentioned that if such systems exist, what kind of attributes they should posses. The following is a non-definitional list from Koza [1998] for such systems. Although much discussion had been carried out regarding these attributes, only a summary is provided here. As would be seen, much of the properties are inherent in the methods of GP. In the thesis it would be shown that at least around 10 of the attributes could be shown in the experiments, although the GP community claims that at least 13 are apparent. This is of course dependent on how far our experiments go towards reconciling an automatic program-creating system, but it should be pointed out that we simply just did what is necessary to solve our problem, and the simpler it is the better. In brief, the attributes are:

**Attribute No. 1** (Starts with "What needs to be done"): It starts from a high-level statement specifying the requirements of the problem.

**Attribute No. 2** (Tells us "How to do it"): It produces a result in the form of a sequence of steps that can be executed on a computer.

**Attribute No. 3** (Produces a computer program): It produces an entity that can run on a computer.

**Attribute No. 4** (Automatic determination of program size): It has the ability to automatically determine the exact number of steps that must be performed and thus does not require the user to prespecify the size of the solution.

**Attribute No. 5** (Code reuse): It has the ability to automatically organize useful groups of steps so that they can be reused.

**Attribute No. 6** (Parameterized reuse): It has the ability to reuse groups of steps with different instantiations of values (formal parameters or dummy variables).

**Attribute No. 7** (Internal storage): It has the ability to use internal storage in the form of single variables, vectors, matrices, arrays, stacks, queues, lists, relational memory, and other data structures.

**Attribute No. 8** (Iterations, loops, and recursions): It has the ability to implement iterations, loops, and recursions.

**Attribute No. 9** (Self-organization of hierarchies): It has the ability to automatically organize groups of steps into a hierarchy.

**Attribute No. 10** (Automatic determination of program architecture): It has the ability to automatically determine whether to employ subroutines, iterations, loops, recursions, and internal storage, and the number of arguments possessed by each subroutine, iteration, loop, recursion.

**Attribute No. 11** (Wide range of programming constructs): It has the ability to implement analogs of the programming constructs that human computer programmers find useful, including macros, libraries, typing, pointers, conditional operations, logical functions, integer functions, floating-point functions, complex-valued functions, multiple inputs, multiple outputs, and machine code instructions.

**Attribute No. 12** (Well-defined): It operates in a well-defined way. It unmistakably distinguishes between what the user must provide and what the system delivers.

**Attribute No. 13** (Problem-independent): It is problem-independent in the sense that the user does not have to modify the system's executable steps for each new problem.

**Attribute No. 14** (Wide applicability): It produces a satisfactory solution to a wide variety of problems from many different fields.

**Attribute No. 15** (Scalability): It scales well to larger versions of the same problem.

**Attribute No. 16** (Competitive with human-produced results): It produces results that are competitive with those produced by human programmers, engineers, mathematicians, and designers.

Attribute No.16 is especially important because it reminds us that the ultimate goal of a system for automatically creating computer programs is to produce useful programs - not merely programs that solve “toy” or ”proof of principle” problems.

# Chapter 4

# Genetic Programming in Control Engineering

The attractiveness in Genetic Programming lies in its possible ability to provide an automated solution without human intervention, combined with the expected robustness of Genetic Algorithms. Further more, it is theoretically capable of using all the existing computing tools and methods used by a programmer to simplify or change the representation of the problem to become solvable, thereby provide some form of optimality and reusability. These are certainly big claims although the community admits that the GP could not solve all the problems.

Hence it would be interesting to see, given the claims on its possibilities, if Genetic Programming could be applied to the field of control engineering. Koza [1999] had demonstrated using GP a time-optimal robot controller with the goal being to find a strategy for continuously specifying the direction for an object moving a constant speed (with a nonzero turning radius) to an arbitrary destination point. If the object has nonzero turning radius, this problem cannot be solved by greedily reducing the distance between the robot and the destination at every intermediate point along the robot's trajectory. Instead, temporal disadvantage might be useful for long-term goal. This is also seen in later sections also where a RST controller is evolved.

Hence what does this tell us? Many ways. GA has proved to be successful in many numerical functions optimisations, scheduling problems [Davis, 1991, Chu and Beasley, 1995] , evolving neural networks [Nolfi and al, 1994] and other control problems [ K. J. Hunt, A. J. Chipperfield an al, 1992 ]. Can GP do the same?

## 4.1 Optimisation

Genetic Algorithms and in fact Evolutionary Computing in general have shown good capabilities in optimisation problems. GP as direct cousins should do the same, if not more, since theoretically it is able to incorporate functions and computing tools. This is going to be demonstrated in subsequent chapters, as reflected in the Lyapunov solver problem and Model Reference Adaptive Systems.

## 4.2 Robust and Adaptive Systems

Nature certainly does it better, so can we mimic such capabilities in problem solving?

One of the sought-after goals in control engineering is to make systems adaptive to changes in the environment. Adaptive controllers have been successful and are available in the market. The role of genetic programming could then be to provide a further tool and complement the existing arsenal of algorithms. Where systems require something rather than greedy 'hill-climbing' methods, GP may provide the 'stop and reverse the damage' property.

Also, structure of GP could probably be used as a background to real-time control of hybrid systems, that is, one single best adaptive controller is doing its job on a particular plant while there may be a whole population of adaptive algorithms that is evolving at the background. Hence at each time the best controller would be used for controlling the plant even when there's a change in the plant process. Adaptive GP controllers are also explored in this thesis, but in the sense of evolution of a population of controllers trying to survive in the changing environment.

## 4.3 Nonlinear Control and System Modelling

Since genetic programming has shown itself capable of evolving solutions such as following a curve and simple regression problems. Perhaps it might be useful to be used as a tool for inverse modeling of non-linear systems, or using internal model methods.

# Part II

# Part II : Experiments

# Chapter 5

# Problem formulation and Structure Representation

As with any other methods, Genetic Programming requires the problem to be set up such that the system is able to interpret and solve it. The structure of tree-like representation of data shown in section 3.6.2 is easy to follow. However it is not trivial to be implemented in computer-languages such as Matlab since the language do not provide pointers, linked-lists, and abstract data types features that are available in other languages such as C. Hence the first job is to provide some properties of pointers and linked lists to our Matlab program. Secondly, such representations must be able to give the method of GP an easy access to alter the structure of the individual programs within the population, since genetic operations such as crossover and mutation would be carried out.

In this thesis three forms of data representations were designed and implemented. Incidentally they can be seen also as 'evolution' over the course of working on the project, as improvements were made and some initial designs were dropped. Nonetheless, in essence, all these data architectures are still the same representation of the tree-like functions structures but only with slight variations. Data Structure I represent tree-like structures in its most general form, while Data Structure II is a more specialised case which is easier to implement than the former. Data Structure IIIA and IIIB resembles the vector strings of the individuals in genetic algorithms.

## 5.1 Data Structure I

What kind of data structure is suitable for implementation in Matlab environment, yet able to provide the versatility for genetic operations? It is proposed that the tree-like structure of functions be 'flatten' into a linear string of codes which can then be represented in Matlab as a row of vectors. This is very much like the binary strings used in genetic algorithms discussed in the previous sections, but instead of binary digits, we have indices to functions and matrices. Therefore individuals in this case would be represented by a row of vectors, and an element in the row can refer to either an index to function such as plus (+) or (-), or the index to a particular matrix. We look at this at more details.

### 5.1.1 Functions

We would like to represent the tree-like structure of a function into the form of a ‘flattened’ linear structure. This means some form of transformation, see Fig 5.1 top half section. A complete function in our ‘flattened’ form can be represented by 4 elements in a row. The 1st element would be the 1st argument it takes in, the 2nd is an index to function, 3 rd is the function’s 2nd argument, and the 4th element is the output of the function. While 2nd element points to the index of the function that is to be performed, 1st, 3rd and 4th element points to different matrices that are stored in the matrix space. A function such as this would then take in 2 arguments from this matrix space, perform its operation, and store it back into the designated space.

### 5.1.2 Matrix Space

In order for Matlab-based GP program to be able to work on any size of matrix, yet able to be represented by a single index properly, we represent the matrices which the functions perform upon all in row form. That is, we transform for example a 2x2 matrix into a row matrix of size 1x4 instead when we stored it in our huge pool of matrix space. In this case, particular matrix in the pool could be called upon easily by simply referring to the row index in the pool itself. See Fig 5.1, lower half.

### 5.1.3 Individual

An individual would then be represented by a string of functions where each function points to different matrices in the matrix space and act upon them. Genetic operations can now be performed on the individuals by working on parts of these strings. See Fig 5.2

### 5.1.4 Evaluation of Individual

When applying the fitness criteria to judge how far the particular individual program is from the actual solution, or if it is moving towards the right direction, usually a test is performed. Typically this will be a run-through of mock variables and arguments. The evaluation program executing these tests is particularly important, as it implements the fitness criteria which in turn is used to judge if the solution is reached, or if not, drives the population towards deriving the right answer. Also, since in most cases, evaluation of individuals takes up most of the computational time, it is essential that evaluation programs written for GP are concise and efficient.

Since the data structure of each individual used in our experiments is linear. It is possible for our evaluation program to run through one individual in a single row, from beginning till the end sequentially and calculates the evolved solution. What the program does is call the first function that appears in the individual, returns the correct value, carries on and call the next function which is directly laid besides the one before, and so on. Thus there is no need for the jumping of pointers to different memory locations or addresses, except when referring to indices of the matrices or functions, thus simplifying the evaluation process.

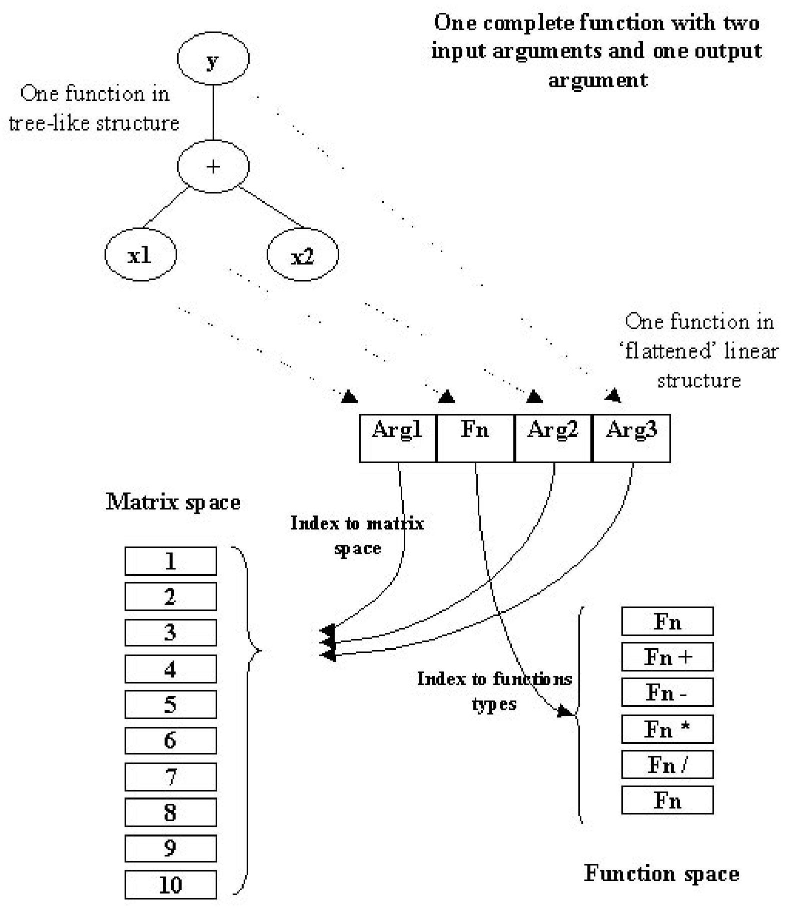

One complete function with two input arguments and one output argument
y
One function in tree-like structure
+
x1
x2
One function in 'flattened' linear structure
Arg1
Fn
Arg2
Arg3
Matrix space
Index to matrix space
1
2
3
4
5
6
7
8
9
10
Index to functions types
Fn
Fn +
Fn -
Fn *
Fn /
Fn
Function space

Figure 5.1: transforming of a tree-like to a 'flattened' linear structure

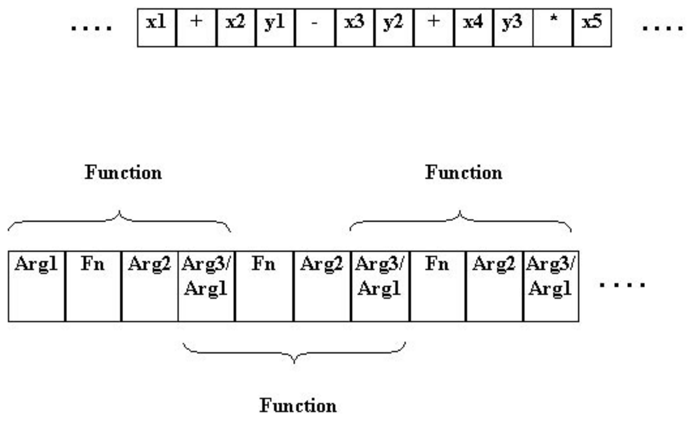


Figure 5.2: An example of an Individual programing linear structure

### 5.1.5 Genetic Operations

The three main genetic operations, as mentioned, are crossover, reproduction and mutation. We shall now see how these are performed on the designed data structures.

**Crossover:** Similar to the binary code example shown in the section 2.1 of Genetic Algorithm, the 'genetic materials' of our individuals, which are now strings of functions instead, are being used in the operation. During a crossover, two individuals are selected based on fitness. Two random but valid points are the chosen on each individual, and this constitutes to one string fragment of genetic material for each. The string fragment of the first parent is inserted into 2nd parent and vice versa, with the original fragments of the parents being replaced. The products are two offspring each carrying the exchanged and combined materials of their parents. See Fig 5.3.

**Reproduction:** One individual is selected from the population based on fitness, and inserted into the population of the new generation.

**Mutation:** One individual is selected from the population based on fitness (there is of course a choice not to choose the fittest ones). Fragments of the string of functions of the particular individuals are deleted and replaced by a randomly generated string instead. Fig 5.4 shows the case where only one random matrix is inserted into the particular individual.

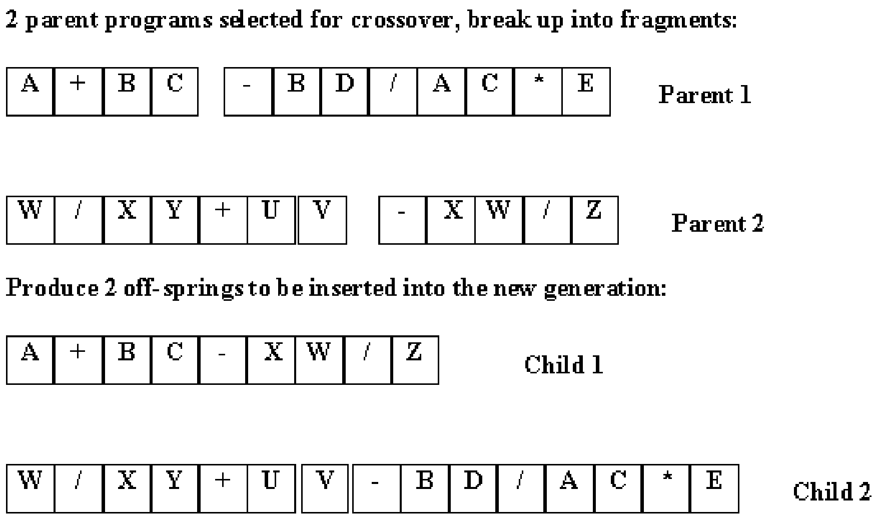

2 parent programs selected for crossover, break up into fragments:
A + B C - B D / A C * E
Parent 1
W / X Y + U V - X W / Z
Parent 2
Produce 2 off-springs to be inserted into the new generation:
A + B C - X W / Z
Child 1
W / X Y + U V - B D / A C * E
Child 2

Figure 5.3: crossover operation done by Data Structure I

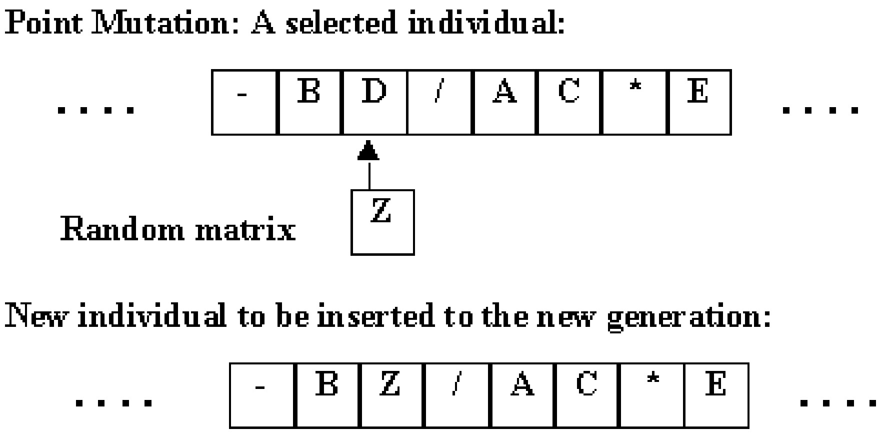

Point Mutation: A selected individual:
. . . . - B D / A C * E . . . .
Random matrix
Z
New individual to be inserted to the new generation:
. . . . - B Z / A C * E . . . .

Figure 5.4: mutation operation done by Data Structure I

### 5.1.6 Architecture Altering Operations

Although such data structures simplified the process of genetic operations, its main disadvantage lies in that it would be more difficult to represent a more complicated multi-tree program. What the current design can represent is only a linear string of functions, (Fig5.2, which is different from that of a branched tree network.

While preserving the existing form of the evaluation program, a crude form of Architecture Altering Operation, argument duplication (perhaps inappropriately named) was implemented. Say we would want to represent $((X+Y)*(X-Y))$. In our normal representation, the evaluation program would interpret the given individual as $((X+Y)*X)-Y)$, that is, without any respect to the brackets and order of operations. Argument duplication in our case is to duplicate $(X-Y)$ first, say equivalent to $(X-Y)=Z$, then performs $(X+Y)\times Z$. Since $(X-Y)$ is now replaced by Z, structure of the individual is changed. Of course the sacrifice is that genetic operations cannot be performed on the branch $(X-Y)$, however the structure of the individual is now again linear, and the evaluation program can be performed.

### 5.1.7 Disadvantages and Advantages

The main problem of such data representation is that conversion between a normal matrix and a row vector is necessary every time when the matrix is used or stored. However, indexing of matrices became easy. Unlike a linked list or a record in other computer languages, a matrix can only take in numbers as its elements. Hence our representation of a function could only contain integers of indices, and as such, there is no clear distinction between the index to matrix and index to function itself. The evaluation program in this case therefore needs to distinguish the two. However the advantages outweigh most of the disadvantages in the sense that once the functions are set up in strings within the individual, evaluation is straightforward, and genetic operations can be performed fairly easily. Also, using indices to matrices allows the matrices themselves to be able to take any size and form. Hence even though most of the time our experiments use only 2x2 matrices, larger-sized matrices are possible.

Data structure I was used in experiments in section 6.1, 6.2

## 5.2 Data Structure II

In our quest for improvement, it was realised that the program structure can do better than that. The main point is that the string of functions in each individual need not have the output values occupy one element space for each function all the time. In other words, since calculation of the functions in the individuals follows from one to the next, only a temporary variable that takes in the value calculated so far (along the string) is needed. See Fig 5.5 lower part. Hence the data structure of an individual can be now represented as seen in Fig 5.5 top part . As seen, only a string with the pattern of indices of functions and matrices in alternate positions is needed.

In this format, evaluation can be carried out at faster speed since the length of each individual can be reduced without compromising the data. Instead of four elements to a function, there are now only two.

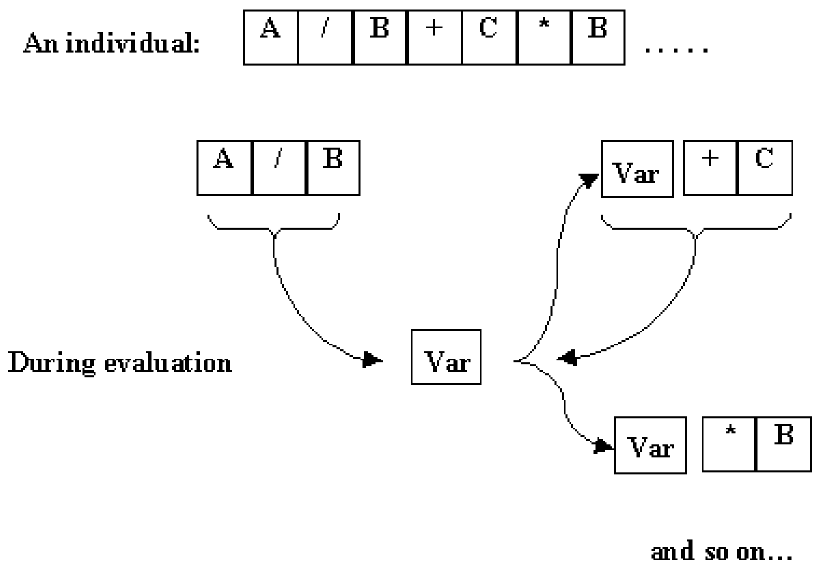


Figure 5.5: Data structure II: reducing the matrix space using temporary variable

Data Structure II was used in experiment in section 6.3

## 5.3 Data Structure IIIA

In genetic programming, it is desirable that the functions of the individuals are able to evolve by themselves during the GP run. See section 3.8 on Automatically Defined Functions. These ADFs provide the features of code re-use, and also breaking down complicated problems into individual smaller solvable blocks. To enable such property, a more advanced data structure was needed. The new design is as follows. In some sense, we have lifted the original data structure II up by one level.

A pool of ADFs is now present in the program population, initialised and able to evolve. They are similar to that of the individuals of data structure II, only that now the original individuals become defined functions or macros instead. The ADFs have a fitness function and fitness values of their own. Genetic operations on ADFs should be of similar methods as that performed on the original individuals. See Fig 5.6.

In the case of data structure IIIA now, the new individuals consists solely only indices of these defined functions. One considerations though, is that during the evaluation of individuals, the program must be able to track where each of the ADFs return the output matrices to, by referring to the correct indices.

Data Structure IIIA was used in experiment in section 7.2. However, only the pre-determined fixed type of functions are reported in this thesis.

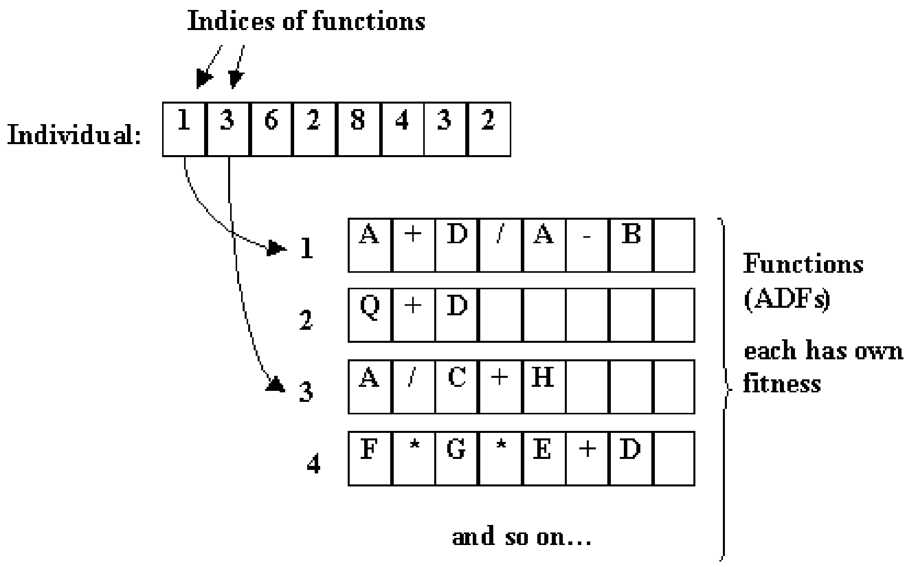


Figure 5.6: Data structure III: Individuals defined by strings of ADFs

## 5.4 Data Structure IIIB

Until then, all the matrices are generated within the programs themselves. They can be either being internally generated numbers or output values of functions. In order for a program to be any good, let alone an automatically generated one, the program should be able to take in both variables and constants defined by the users. Data structure provided this feature by letting functions takes on global variables within the program. See Fig 5.7.

In the later part of our experiments, evolving ADFs were not needed to produce satisfactory results. Hence functions used were non-evolvable type. In fact, in data structure IIIB, ADF pool was not used at all. It appears that the simpler solution to representing data is the best and we had come full circle.

By giving individual functions control of global variables, tracking of indices of matrices is no longer that complex as before. What the program would do instead now, is to track the declared indices of variables and constants only, rather than the huge matrix space presented before. Individuals are represented as like Data Structure IIIA, consisting of string of indices to functions. Functions are no longer automatically generated, but are now provided by the user.

However, the main advantage would be that more complicated multi-tree structure could be represented fully, unlike those of earlier versions. The user would able to decide the number of variables each individual function would be able to take in, and also the number of output arguments. The capability of multi-input multi-output functions is provided.

The disadvantage now would be that the user who designs the function operators would need to manually take caution on the passing of variables and constants within each written function. Also, the user need to have some intuition on the problem itself and provide what he/she thinks might be useful in helping to evolve a useful program.

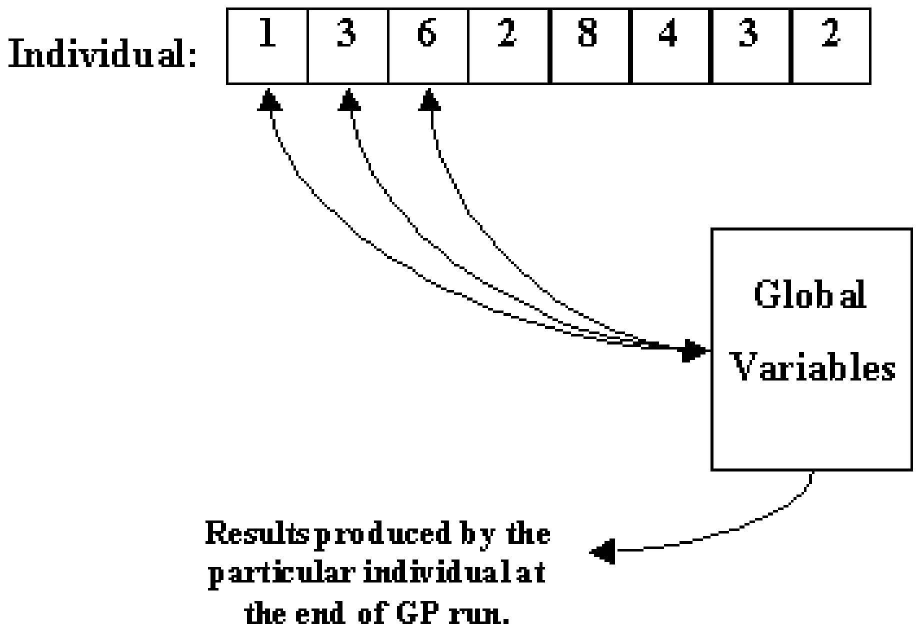

Individual:
1
3
6
2
8
4
3
2
Global Variables
Results produced by the particular individual at the end of GP run.

Figure 5.7: Data structure III: Introduction of global variables for input/output

Data Structurer IIIB was used in experiment in section 7.1

# Chapter 6

# Simple Experiments

## 6.1 Simple Demonstration of Convergence

The first problem is of course to make sure that if there is a solution, the program will converge to it.

An initial population of individual programs containing of one or two functions was created. During a GP run, two kinds of operations were performed according to pre-set probabilities. Firstly is of course the required genetic operation. Secondly, a particular individual can increase or decrease its string length. This was done by having random matrices being added or subtracted to the original individual. This operation was to allow increased varieties within the population in the probable cases where solutions (or near solutions) could not be found or choices were exhausted.

Unlike the usual methods of genetic algorithms, where vectors or binary codes are exchanged between parents in genetic operations, sets of random matrices, which are produced together with the functions, are once generated and cannot be changed. That is, genetic operations are not allowed to act on the elements of the matrices. Instead, in this GP experiment, genetic operations are restricted to the exchange of strings of set functions. Therefore, the run of GP is a search for sets of created functions (together with the matrices) to reach the required goal. It would then be in this case an optimisation of both the available matrices and the functions. It would be clearer with the following illustrated example. The data structure for experiment 1 is of type I.

A 2x2 matrix, say [1 0; 2 -1] was selected. The goal is to have an individual match as close as possible to the selected matrix. The fitness criteria would be the sum of absolute error between the generated matrix and the desired matrix.

### 6.1.1 Parameters

Fitness Criteria: Sum of absolute error of the elements between generated matrix and desired matrix

Parameters: Generations: 10

Population: 120

Functions: addition, subtraction

### 6.1.2 Results and Discussion

The program was able to converge to within 15% of accuracy almost all the time. Of course this would also be dependent on the available generated random numbers.

Since the fitness criterion was the sum of the absolute error, it came as no surprise that the program did not judge how close each element of the matrix to the desired one. Hence it may generate a matrix that is very accurate in perhaps 1 or 2 elements, but inaccurate in the others.

Finally, it is unclear if the size of population per generation affects the rate of convergence. We would expect because the higher the size of population the higher the probability that there are matrices that fulfil the fitness criteria. However, a larger size also means a wider variety of combinations.

Generally the rate of convergence and accuracy are directly influenced by how tough was the fitness criteria. The tougher the fitness criteria, i.e. the best matrices in the generation were given a much higher fitness points than others, the faster GP convergence to the rough solution. However the best results will not be as accurate and vice versa. Hence the rate of convergence is probably inversely proportional to accuracy of the solution.

## 6.2 Simple Lyapunov Solver

### 6.2.1 Background of the problem

Lyapunov theory has been widely used in control engineering for the testing of stability of systems. We now provide a short background and built-up to the problem that was presented for genetic programming. Details of Lyapunov methods can be found in standard text such as Khalil, 1997, or Slotine, 1991.

Lyapunov method is the mathematical extension of a fundamental physical observation that if the total energy of an electrical, or mechanical, systems is continuously dissipated, then there must exist some equilibrium point where the particular system will settle down, regardless whether the system is linear or non-linear. Hence, we may conclude the stability of a system by examining the variation of a single scalar function.

Readers who are familar in this area may like to jump directly to discussions on the experiment in section 6.2.2. Otherwise, the following section gives some background on the problem which we are going to solve. In later part, section 7.1, we would refer back to this section again.

The following we make a few definitions.

#### Non-linear Systems

Usually non-linear dynamic systems can be represented by a set of non-linear differential equations in the form

$$\dot{x} = f(x, t)$$

Where $f$ is a non-linear vector function and $x$ is the state vector. When such systems have $f$ that does not depend explicitly on time, that is,

**Equation 1**

$$\dot{x} = f(x)$$

The systems is said to be $autonomous$. Otherwise, it is called $non-autonomous$.

#### Positive definite and semi-definitive functions

A scalar continuous function which is differentiable $V : R^n \to R$ is called $positive - definite$ in a region$U \in Rn$ containing the origin if

1. $V(0) = 0$
2. $V(x) > 0, x \neq 0$, where $x \in U$

A function is $positive - semidefinite$ if condition 2 is instead, $V(x) >= 0$.

#### Stability

With $x$ denoting the state of the system, the system is said to be *stable* if, for any $R > 0$, there exists $r > 0$, such that if $\|x(0)\| < r$, then $\|x(t)\| < R$ for all $t >= 0$. Otherwise the equilibrium point is *unstable*. The solution is asymptotically stable if it is stable and $r$ can be found such that all solutions with $\|x(0)\| < r$ have the property $\|x(t)\| \to 0$ as $t \to \infty$.

## Lyapunov function and Lyapunov Stability Theorem

Now with the autonomous system in Eq. 1, the scalar function $V(x)$ actually represents an implicit function of time $t$. If such function $V(x)$ exists such that we can take its derivative with respect to time and applying chain rule,[4].

**Equation 2**

$$\dot{V} = \frac{dV}{dt} = \frac{dV^T}{dx}\frac{dx}{dt} = \frac{dV}{dx}f(x)$$

We can now define the following: if $V(x)$ is positive definite and has continuous partial derivatives, plus if its time derivative along any state trajectory of the system is negative semi-definite, i.e.

$$\dot{V} <= 0$$

then $V(x)$ is said to be a Lyapunov function for the system. Furthermore, we can conclude that the solution $x(t) = 0$ to the system is stable. If $\dot{V}$ is negative definite,

$$\dot{V} < 0$$

The solution is also *asymptotically* stable. In addition, if it is asymptotically stable, and $V(x) \to \infty$ when $\|x\| \to \infty$, The solution is *globally* asymptotically stable.

## Lyapunov Functions for Linear Time-Invariant Systems

Given a linear system of the form $\dot{x} = Ax$, consider a quadratic Lypunov function candidate

**Equation 3**

$$V = x^T P x$$

Where $P$ is a given symmetric positive definite matrix. Differentiating the positive definite function $V$ along the system trajectory yields another quadratic form

$$\dot{V} = \dot{x}^T P x + x^T P \dot{x}$$

**Equation 4**

$$\dot{V} = x^T A^T P x + x^T P A x = -x^T Q x$$

where $A^T P + PA = -Q$ is called Lyapunov equation.
Thus if the symmetric matrix $Q$ is positive definite, it means that $V$ satisfies the conditions defined as before and the system is asymptotically stable. A proof of it would be presented in later section (7.1). However, it should be noted that $Q$ may not be positive definite even for stable systems.

A better approach would be instead, to derive a positive definite matrix $P$ from a given positive definite matrix $Q$, that is,

1. choose a positive definite matrix $Q$
2. solve for $P$ from the Lyapunov equation (4)
3. check if $P$ is positive definite.

If $P$ is positive definite, then $V = x^T P x$ is the Lyapunov function for the linear system and global asymptotical stability is guaranteed.

### 6.2.2 Experiment

As can be seen, Lyapunov theory addresses an important aspect in control engineering. Our GP problem then was to solve for $P$ from a given a stable A and positive definite $Q$. Our approach was similar to that of problem 1, with data structure type I. However, the fitness criteria for this case were of course different. We would like our GP system to besides solving the Lyapunov equation in Eq.4, it has to ensure that $P$ is also positive definite. Some of the properties of a symmetric 2x2 matrix (in the case which we were solving) which is positive definite are:

$$P_{11} > 0$$

$$P_{22} > 0$$

and

$$P_{11}P_{22} - P_{12}^2 > 0$$

This is simply to ensure that $P$ has positive determinants. Nonetheless it passed as suitable fitness criteria.

Other than that, the rest of the program is the same as that of previous problem.

#### Parameters

Fitness Criteria: Absolute error between $Q*$ and the pre-determined $Q$, where $Q*$ is the output value of the Lyapunov equation using the evolved solution $P$, i.e. $A^TP + PA = -Q*$. Also, a higher reward is given to the particular individual if the generated $P$ can satisfy conditions stated 6.2.2.

Generations: 10

Population: 80

Functions: addition, subtraction, multiplication, division

### 6.2.3 Results and Discussion

As like before, we saw that given the randomly generated matrices and mathematical operators, our GP solution was able to generate results $Q^*$ that were fairly close to the pre-determined $Q$ while at the same time having $P$ satisfy the given conditions. A run of our GP system would typically look like this:

```
gpmain

Avgfit =

  611.7195

one generation done

Avgfit =

  312.1390
```

```
one generation done

Avgfit =

  1.8629e+003


QQ =

   -0.9080   -0.1580
   -0.9300   -1.5120

one generation done

Avgfit =

  1.4132e+003


QQ =

   -1.0750    0.0596
   -0.7085   -0.5143

one generation done

Avgfit =

  1.4186e+003


QQ =

   -1.0383   -0.3693
   -0.9811   -0.2673

one generation done
GP run stopped, fitness met

Qtest =

   -0.9100    0.0750
   -0.9861   -1.1014


P =

    1.0057    0.0201
    1.0813    1.1008
```

```
Avgfit =

  4.2011e+003


QQ =

   -0.9100    0.0750
   -0.9861   -1.1014

one generation done

bestfit =

  2.2606e+005


bestfitresult =

    1.0057    0.0201    1.0813    1.1008
```

This particular run results in $P$ matrix which is satisfactory and the GP program exits since the termination criteria have been met. The solution of GP is a combination of various mathematical operations and randomly generated numbers, and is shown below: [1]

```
Bestresults =

((([0.269 0.821 ; 0.751 0.914 ]*[0.0509 0.593 ; 0.732 0.332 ])-[0.25
0.799 ; 0.334 0.426 ])+[0.641 0.387 ; 0.708 0.778 ])
```

[1]the 2-by-2 matrices have been represented in row format for easy presentation on the computer

## 6.3 Optimal Control

We now solve a simple optimal control problem using our GP system. The normal method of solving the problem is first presented and then the method using genetic programming comes next.

### 6.3.1 Background of the Problem

Assume an athlete is going to take part in a 100m race. However he has only a limited amount of energy to spare, and so he need to find a suitable strategy to complete the race in minimum time to maximise his chance of wining.

Mathematically, this problem can be written as

**Equation 5**

$$min \int_0^{t_f} 1dt$$

with constraint on energy

$$\int_0^{t_f} u^2 <= w_o,$$

say with $w_o = 100$.
Initial conditions are:

$$x(t) = 0$$

where $x$ represents the distance covered, and

$$\text{Velocity} = 0$$

$$\text{Consumed energy} = 0$$

Final condition is

$$x(t_f) = 100$$

where $t_f$ is the final time
We can write the equations in state space representation

$$x_1 = x$$

$$x_2 = \dot{x} = \text{velocity}$$

$$\dot{x_2} = \text{acceleration}$$

Energy constraint can be written as

$$x_3(t_f) <= w_o \text{where}$$

$$\dot{x_3} = u^2$$

We see that the system is of the form $\dot{x} = f(x, u)$ and the minimising equation is in the gerneral form:

$$min \int_0^{t_f} L(x(t), u(t))dt + \phi(x(t_f))$$

where $u(t) \in U$ is the control signal in time $t$.

Typically to solve this problem, we use *The Maximum Principle* (see [[12]]) by firstly introducing the *Hamiltonian equations* defined by

**Equation 6**

$$H(x,u,\lambda) = L(x,u) + \lambda^T f(x,u)$$

Suppose there is a solution $u^*(t), x^*(t)$ then the optimal solution must satisfy

$$minH(x(t),u,\lambda(t)) = H(x^*(t),u(t),\lambda(t)), 0 <= t <= t_f,$$

where $\lambda(t)$, called the *Lagrange multiplier*, solves the adjoint equation:

$$\dot{\lambda}(t) = -H_x^T(x(t),u,\lambda), \lambda(t_f) = \phi^T(x^*(t_f))$$

We now rewrite the final state conditions at $t_f$,

$$x_1(t_f) = 0$$

$$x_2(t_f) = 1$$

$$x_3(t_f) = w_o$$

$$t_f = free$$

The Hamiltonian equation in our case is

$$H = \lambda_1 \dot{x_1} + \lambda_2 \dot{x_2} + \lambda_3 \dot{x_3}$$

$$= \lambda_1 x_2 + \lambda_2 u + \lambda_3 u_2$$

differentiating w.r.t $u$,

$$H_u = \lambda_2(t) + 2\lambda_3 u(t) = 0$$

The adjoint equations will be

$$\dot{\lambda_1} = -H_{x1} = 0$$

$$\dot{\lambda_2} = -H_{x2} = -\lambda_2$$

$$\dot{\lambda_3} = -H_{x3} = 0$$

Then

$$\lambda_1 = \sigma_1(\text{free})$$

$$\lambda_2 = \sigma_2(\text{free})$$

$$\lambda_3 = \sigma_3(\text{free})$$

are all free constants. Solution of $u$ from $H_u$ is then of the form

$$u = A - Bt$$

which is a slope with negative gradient. We will not go into further calculations in detail other than that. However, this is enough to say that, our athlete should apply maximum strength in the beginning for acceleration, and then trails off gradually till the ending point is reached.

### 6.3.2 Experiment

We now use genetic programming to approach the problem.

A pool of random vectors each of size (1 x 30) representing control signal $u$ at each time $t$, is generated. This signifies the random strategies that can be used by the athlete across a time period that is more than the required optimal time to complete the race. That is, if the optimal $t_f$ can be achieved is 10s, we put the total time to apply $u$ to say, 30s. Now our GP system is required to use these random vectors and combine using mathematical operations to find a right strategy such that call the conditions are fulfilled.

The fitness criteria used in this case is of course the returned time $t_f$, time to complete the run, produced by the evolved solutions, with constraint exceeding the maximum energy not allowed. The experiment uses data structure II described in section 5.2.

#### Parameters

Fitness Criteria: time $t_f$ to reach the end-point, total energy limited to 100

Parameters: Generations: 10

Population: 100

Functions: addition, subtraction

### 6.3.3 Results and Discussion

As can be seen, (Figure 6.1) the program returned a rough but correct answer. As again, GP uses what is available in the limited number of generations to generate the results. Also, it is observed that the total energy used in the evolved solution is near to 100, which signifies that the strategy tries to consume all the available energy. The evolved answer is about 6s, which incidentally breaks the world record.

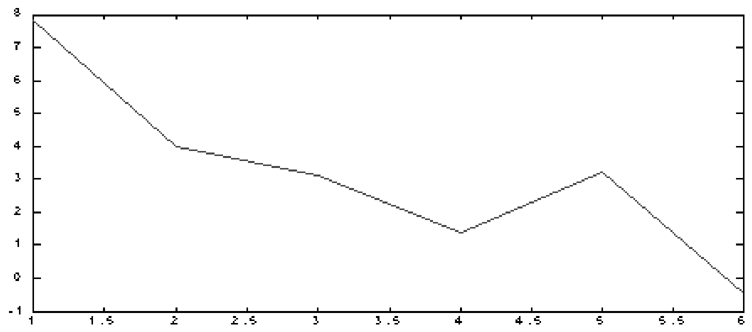


Figure 6.1: Control signal for evolved solution

## 6.4 Average fitness for Problems

An observation was made that over the generations, the average fitness of the population increased generally over time. However, there may be some instances when the average fitness actually decreased. This should not be surprising since in genetic algorithms it does not necessary mean that for every genetic operations performed on an individual, the new offspring would be better than its parents. It is also because of this property, GP search is not a greedy hill-climbing one.

## 6.5 Conclusion

In the previous section we presented three simple problems, two of which are related to control engineering. It could be seen that GP optimises both the usage of its given functions and the generated matrices. There is also no explicit mathematical calculation done by the system, only genetic operations. Whereas if we solve the problems by hand, they will typically take some effort. Finally, the percentage of cross over operations is 75%, and reproduction is 25%. This would be used throughout all the experiments except in Section [7.2].

# Chapter 7

# Advanced Experiments

## 7.1 Discrete Time Lyapunov Solver

Up till now problems solved by genetic programming can be considered to some being trivial or 'toy problems'. For a system with the capability to automatically create programs, it is necessary to show that the system is able to go beyond that limitations and exhibit some form of intelligience. The following experiment hopes to demostrate that GP, if given the right problem representation is indeed powerful; if not, at least bring it a step nearer towards it. For the first part, we continue to look at Lyapunov functions, but this time, with just a little more detail. As again, readers who are already familar in these topics may like to jump directly to the next section. For those who would like to read further, please refer to [Åstrom, Wittenmark, 1996].

In the previous section we have shown that given a linear systems of the form

$$\frac{dx}{dt} = Ax$$

where it is a asymptotically stable , there exists a Lyapunov function, $V(x) = x^T Px$ that satisfy the equation

**Equation 7**

$$A^T P + PA = -Q$$

We now provide the proof for it.

Let $Q$ be positive definite. Define

$$P(t) = \int_0^t e^{A^T(t-s)} Q e^{A(t-s)} ds$$

The matrix P is symmetric and positive definite because an integral of a positive definite matrices is positive definite too. The matrix $P$ also satisfies

$$\frac{dP}{dt} = A^T P + PA + Q$$

Since the matrix $A$ is stable, the limit

$$P_o = \lim_{t\to\infty} P(t)$$

exists, satisfying Eq. 7. It can also shown that the solution is unique.

For the case of discrete systems, we can define the linear system as

$$x_{k+1} = Ax_k$$

with

$$V_k(x) = x_k^T P_k x_k,$$

and

**Equation 8**

$$V_{k+1}(x) - V_k(x) < 0$$

since we know that the gradient of Lyapunov functions is negative.

Now the right hand side of Eq. 7 can be written as

$$x^T P_{k+1} x - x_k^T P_k x_k = x_k^T A^T P_{k+1} A_k - x_k^T P_k x_k$$

$$= x_k^T (A^T P_{k+1} A - P_k) x_k < 0$$

we can see that the equation can then be written as

**Equation 9**

$$A^T P_{k+1} A - P_k + Q < 0$$

which is the Lyapunov equations in discrete form

Similar to that of the continuous time case, there will exist

$$P_o(k) = \lim_{k \to \infty} P(k)$$

If the linear system is asymptotically stable. This would mean that at some discrete time $k$, $P$ will settle down to a unique value.

It is not difficult to see that inherently that this equation contains 2 main features. They are 1) iterations involved, 2) some kind of memory is needed. A system that can solve this equation thus requires some kind of memory allocation and also able to do iterations of some form. This proved to be a good testing ground then for our GP system to automatically conduct program discovery, for one it should also possess these two properties. A further requirement is that the evolved solution should also be able to take in any given $A$ (stable) and $Q$ (positive definite), solve the Lyapunov equation and return the correct value $P$. For this we look at two tools in genetic programming which might be useful for our experiments.

### 7.1.1 Automatically Defined Loop

Automatically Defined Loops (ADLs), as defined by Koza, provides the mechanism by for genetic programming to implement a general form of iteration involving an initialisation step, a termination condition, a loop body and an update step. It consists of four distinct branches. That is,

1. a loop initialising branch

2. a loop condition branch

3. a loop body branch

4. a loop update branch

These are all subjected to evolutionary processes during the run of GP. During the run, if the loop is called, a fixed structure (pre-determined an not subjected to modification) causes the loop to be initiated. The condition branch will see if the loop should be continued or terminated based on whether the condition specified is met. If not, the body of the loop is executed, and the loop is updated. The loop will terminate as soon as the condition branch returns a signal to do so. This sort of ADL in its terminology is similar to the FOR loop used in many computing languages such as C. Please see [Koza, 1999] for details.

In our case, the ADL structure was not exactly followed. Although it can be seen that iteration should be necessary to solve the Discrete Lyapunov Equation problem, such advanced feature like ADL was in fact not needed. In fact, it turned out that a simpler method, perhaps more elegant, was implemented. That is, to allow extra function spaces within individuals to be used if necessary. During the GP run, the evolutionary process will decide whether it should be used, and if so, whether it should be in an iterative manner. More would be discussed in the section 7.1.3.

### 7.1.2 Automatically Defined Storage

Internal storage (memory) is convenient, and often necessary in the creation of computer programs. However, often it may not be obvious as to whether a given problem requires memory, and if so, how much storage space is needed. Even if the amount, type and dimensionality of internal memory are known, it is not trivial to decide what exactly is needed to store in memory and what to retrieve. It is therefore desirable, to have a system that can automatically make such decisions and also specify the way in which particular types of memory to be used.

In [Koza,1999] also, Automatically Defined Storage (ADS) is the mechanism for GP to implement the general form of internal storage. In brief, this is done by adding 2 new branches to the given computer program.

1. a storage writing branch

2. a storage reading branch

The storage writing branch may be seen simply as a WRITE statement while the storage reading branch can be seen as a READ function. The pair of branches of an automatically defined storage by themselves is not used for keeping any executable code. Rather, when internal storage is invoked (either added or taken away from), it provides an administratively convenient way to expand (or contract) the program's function set. Some points which we need to consider is the type of ADS which is generated, and the dimensionality of the ADSs, defined as the number of arguments that is need to address the storage space. [Koza, 1999].

From the Discrete Time Lyapunov equation it is shown also explicitly that internal storage is necessary, that is, for storage of variables $P_k$ and $P_{k+1}$. It is not hard to see also, that the automatically created program may want to store the results of $A^T PA$ for future use too. In order not to complicate the problem, and also force the generated solution to make the right choice of storage allocation all the time, only two variable memory spaces were provided. This storage space is able to take in matrices of sizes defined in the beginning of the problem. There are also two constants, namely the $A$ and $Q$ matrices, that are to be defined by the user.

### 7.1.3 Experiment

The genetic programming solver for this problem is of data structure type IIIB. Individuals consist of only strings of functions. Since this is the case, the moving of generated results to and from the functions or individuals lies in the predefined functions themselves. That is, as different from previous cases where functions are automatically defined and passing of arguments lie within the individuals, each function in this GP system passes results through global variables. In short, each function simply performs its operations in a defined commonly shared space.

From Fig. 5.7 in Section. 5.4 we see that each function retrieve and acts on the shared variables.

Design of the function operators that are suitable for solving this problem (and of course also for each particular problem) are crucial. As [Kinnear, Jr, 1994] advises, "always pick the most powerful and useful seemingly functions from the problem domain that you can think of". [Langdon,1999]

In our experiment, 16 pre-set functions were created for the GP run. They are of the type:

```
Var1 = Var1 + Q, Var2 = Var1 + Q,
Var1 = Var1*A, Var1 = Var2*A,
etc
```

Where Var is the variable space which the GP system can use. The list of functions is in Section [7.1.3]

#### Initial population and Generation of Individuals

We initialised each individual to a string length of 45 function spaces. This is just a wild guess of the number of function needed to complete the solution. Actual length of the solution should be automatically defined via GP run. In order to achieve satisfactory results to solve the problem, the method of genetic programming requires not only that it chooses the correct functions in the correct manner in each sequence, but also, arrange them in iterative manner if necessary. Hence this is what we meant by not needing an advanced feature of Automatically Defined Loop. If it is needed, given the empty extra function spaces, GP will automatically duplicates the necessary functions and place them in the right order, within the individuals.

**Fitness Criteria**

5 to 7 individual tests were used to judge each particular solution. These tests involve giving a determined $A$ and $Q$ to the evolved individual, judging the outcome $P_k$ from the solution. The generated $P_k$ was substituted into the Discrete Lyapunov equations and absolute residual error calculated, i.e.

$$A^T P_k A - P_k + Q = e$$

where $e$ is the absolute residual error. The sum of absolute residual error generated by all the tests, which the particular solution has undergone, was used as the fitness measure for that solution. Each of these test has various degree of difficulties, that is, some may require more iterations than the rest.

**Parameters**

Fitness Criteria: sum of absolute residual error from all the tests
Generations: 25
Population: 70
Functions:

```
Var1 = Var1 + Q, Var2 = Var1 + Q, Var1 = Var2 + Q,Var2 = Var2 + Q
Var1 = A'Var1, Var2 = A'Var1, Var1 = A'Var2, Var2 = A'Var2
Var1 = Var1*A, Var1 = Var2*A, Var2 = Var1*A, Var2 = Var2*A
Var1 = Var1 - Var2, Var2 = Var2 - Var1, Var1 = Var1 - Var2,
Var2 = Var2 - Var1
```

where Var stands for the global variable.

### 7.1.4 Results

It is observed that a satisfactory solution can be found roughly after 15 generations. A closer examination of the generated program shows that GP arranged automatically the necessary function to the correct order to generate the 'fitter' solutions. Validation tests were made by entering user-defined $A$ and $Q$ matrices to derive the $P$ matrix. Actual iteration tests to get the correct $P$ were conducted and compared with that from the generated program. A typical test after the best solution from the GP run was found looks like this:

note: A stable but random $A$ is generated from matrix $T$ that is random 2-by-2 matrix by the equation $A = T * [0.5 - 0.5; 10] * T'$

```
 Q = [2 0; 0 2];
 results = testresults(bestindividual,Q)
T =
   0.62397334027302   0.43782779805041
   0.77084192127972   0.30845913641846
A =
   0.33126780149269   0.48175328641616
   0.26421467139103   0.41598525047765
```

results of P from evolved solutions

```
GPresults =
   2.78137678518103   1.18084563233129
   1.18084561088413   3.78583973864274
```

number of iterations needed from normal solutions = 6 results of P from iterations

```
Pk =
   2.73577175618569   1.11152076181400
   1.11152076181400   3.68045802326418
```

where both are similar.

The size of the solution was not pre-determined, but GP was able to generate the necessary size for the optimal solution automatically.

It is indeed quite amusing that without any prior knowledge, totally no hint at all that iteration is needed, the solution has arranged its function into the correct manner such that right $P$ would be generated each time. Further tests concluded that the generated results from the program is within less than 5% accuracy each time. The program is also good up to solutions that require 10 iterations.

### 7.1.5 Search Space in the Experiment

We now show the search space that the genetic programming system has to explore to show the validity of the technique. The total probable search space is first presented, and then the probable number of iterations that GP probably took will be discussed.

The number of functions per individual program at the beginning of the GP run was set to 45. This is only guess on the number of required functions that is need to produce the results. There are 16 available pre-set functions in total. The actual solution of course does not need to take all the available functions but rather, based on selection. However if we want to test all combination in an enumerative search the following calculation can be made:

Initial no. of functions in Individual: $I = 45$
Assume the size per individual do not change, which we know is not probable, but for easy calculations we set it to be true. This is also the lower expected limit.
No. of functions available: $f = 16$
We require that the sequence of functions to be in correct order. Hence the search space would be:

$$f^I = 16^{45} = 1.53 \times 10^{54}$$

For a random search, this would be the maximum number of trials and operations it has to go through to reach the correct solution. Consider now the GP case.

We have a population size of 70 individuals, and run of 25 generations. Suppose we loosen the restriction that we can find a solution in 50 generations. This would make

$$\text{Generations x population} = 50 \times 70 = 1750$$

That is, GP requires genetic operations to complete the task, which is significantly less than previous calculations.

There are of course many ways where we can debate over the validity of such analysis. For example, what is the worst case for genetic programming in times of extreme bad luck? Perhaps as bad as random search. What we can do show however, is that there is a higher chance that given such a set of problems, GP might be able to perform better than other algorithms. However, success rate is still not guaranteed to be 100%, at least we do not have to courage to claim that. However, we now provide some intuition on why genetic programming, or in fact genetic algorithms themselves will work in such particular problems.

### 7.1.6 Foundations

This thesis is not a study on the exact working of GA or GP and other search techniques, but is rather, the application of the methods in the area of control engineering. However, it is useful to provide some intuitive or mathematical treatment on why these techniques work or why they will not. Three theorems are grossly summarised below. The Schema Theorem and Price theorem provide support to Genetic Algorithms, whereas the No Free Lunch theorem states that no search techniques is better than the others across the whole probable problem space. From the practical point of view we do not take any sides as long as the method works for our problems. For details of these theorems please refer to bibliography.

#### Schema Theorem

Schema (or schemata in plural) in provides some insight on what exactly is the information, given the payoff data (fitness value) that is inherent in the population of strings which helps guide them towards the goal. Firstly we are looking for similarities among strings in the population, secondly, we are looking for causal relationships between these similarities and high fitness [Goldberg, 1989]. [Holland, 1973] introduced the term schema, a similarity template describing a subset of strings with similarities at certain string positions.

For convenience of discussion, we limit ourselves without loss of generality, a sting of binary code containing alphabet 0 & 1. Say, we consider schemata of length 5 where an example could be 10110. Next we introduce a * or don't care symbol. Now if we think a schema as a pattern matching mechanism, where a schema matches a particular string at every location. This occurs if the schema matches 1 to a 1 in the string, 0 to 0, or a * matches either in the right positions. Hence as example if a schema is *0000, it can matches two strings 10000 or 00000. If the schema is 0*1** it means that it can match any string with length 5 and has 0 in the first position and 1 in the third. Hence where does this lead?

We now consider the genetic operations of reproduction, crossover and mutation on the growth or decay of important schemata over from one generation to the next. The reproduction of a particular schema is easy to determine, since the fitter strings will have a better chance of being selected and inserted into the next generation. Consider now crossover operations, say we have two schemata, 1***0 and **11*. A crossover applies a cut on both schemata, which will disrupt them. Now we can see that the first schema is more likely to be disrupted than the second. The second has the higher chance of not being destroyed. We infer that as a result, schemata with short defining length are left alone by crossover and reproduced at a good sampling rate by reproduction operator. Mutation affects the schemata at low rate since its percentage is small.

Hence we are left with a interesting conclusion, that highly fit, short-defining length schemata are propagated from generations to the next by increasing the samples with the observed best. This goes in parallel without any particular record or special memory.

## Price Theorem on Genetic Algorithms

Price's Selection and Covariance Theorem [Price, 1970] from population genetics relates the change in frequency of a gene within the population from one generation to the next, to the covariance of the frequency of the particular gene in the original population and number of offspring produced by individuals. The theorem holds for a single gene or linear combination of genes within the whole domain, interaction between genes, most kind of species, and any kind of mating. It is particularly applicable to genetic algorithms. [Altenberg, 1994]. We state the theorem below.[Price, 1970].

**Equation 10**

$$\triangle Q = \frac{Cov(z,q)}{\bar{z}}$$

$Q$ = Frequency of given gene, or linear combination of genes, in the population
$\triangle Q$ = change in Q from one generation to the next.
$q_i$ = frequency of gene in the individual $i$.
$z_i$ = number of offspring produced by individual $i$.
$\bar{z}$ = mean number of chil dren produced
Cov = covariance.

When the population size is unchanged, as is usually in the cases of GA and GP,

$$\bar{z} = pr + pm + 2pc$$

where
$pr$ = reproduction rate,
$pm$ = mutation rate,
$pc$ = crossover rate,
note that two parents are required for each individual created by crossover, hence $2pc$ in the equation.

since $pr + pm + pc = 1$, the mean number of children is mean $z = 1 + pc$, Eq. 10 becomes

**Equation 11**

$$Q = \frac{Cov(z,q)}{1 + pc}$$

The implication of this theorem suggests that the covariance between parental fitness and offspring fitness distribution gives the fundamental power to genetic algorithms [Altenberg, 1995]. Further more, [Altenberg 1995] was able to derive Holland's schema theorem [1973, 1992] from Price's Theorem.

**No Free Lunch Theorem**

Whereas the above two theorems suggest the inherent power of genetic algorithms, Wolpert at el [1995] in their No Free Lunch Theorem (NFL) showed that all algorithms that search for an extremum of a cost function perform exactly the same, according to any performance measure, when averaged over all possible cost functions. In particular, if algorithm A outperforms algorithm B on some cost functions, then loosely speaking there must exist exactly as many other functions where B outperforms A. We provide below a brief implications of their results, without any mathematical treatment or formal proof.

One might expect that hill-climbing usually out-performs hill-descending if one's goal is to find a maximum of the cost function. We might also expect that it would outperform random search. However, as the theorem's central results show, this is not the case. If we do not take into account any particular biases or properties of our cost function, then the expected performance of all algorithms on that function are exactly the same (regardless of the performance measure used). In short there's no 'free lunches' for effective optimisation; any algorithm performs only as well as the knowledge concerning the cost function put into the cost algorithm. That is to say, even if one's goal is to find a maximum of the cost function, hill-climbing and hill-descending are equivalent, on average.

There are of course an huge amount of literature and debates over the fruitfulness of search algorithms in particular to what each camps claims. For our works in control engineering, the purpose is to show that genetic algorithms work for your problems, but it is not to say that it will work for all problems. As NFL theorem puts it neatly, it all depends on the bias of your cost functions, and how we trim our problems so that our algorithms would be effective.

### 7.1.7 Conclusion

In this experiment, we have shown that genetic programming is capable of organising, set the arrangement of functions, and provide correct memory allocation without being explicitly being told to do so. The only hint given were the requirements of the end solution, as communicated by the high-level statements to the GP system that is to reduce absolute error of the Lyapunov equation. We have hence shown the following attributes, although not exhaustive, of genetic programming,

1. automatic determination of iteration, if necessary
2. automatic correct usage of memory
3. automatic determination of program size necessary for solution
4. able to reach solution without being told how to do it, but from what is required.

We have also look at various theorems on why genetic algorithms and genetic programming works and why however, we should not be complacent and assume that this method will works for all kinds of problems.

## 7.2 Self-Evolved Model Reference Adaptive Systems

### 7.2.1 Model Reference Adaptive Systems

A Model Reference Adaptive Systems (MRAS) is an important class of adaptive controllers. It is a system where the desired performance is expressed in terms of a reference model, which gives the desired response to a command signal. One of the valuable properties is that it presents a convenient way to give specifications for a controller servo problem. For those who have encountered MRAS for the first time, we provide some background information. For details please see Åstrom and Wittenmark, 1995. Others might like to skip directly to Section 7.2.3.

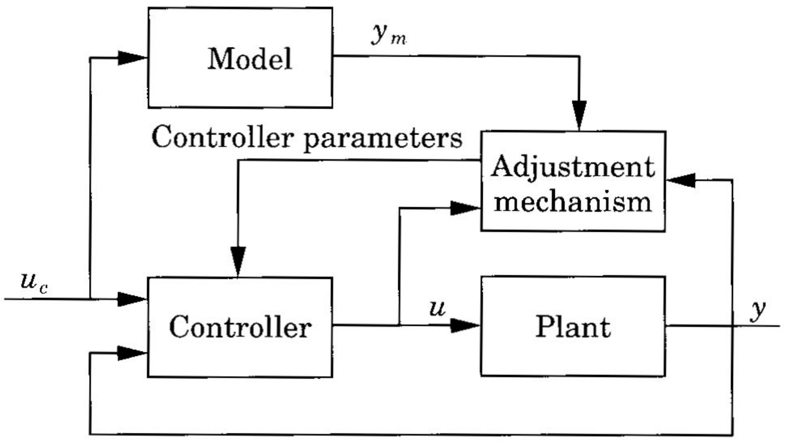


Figure 7.1: Block Diagram of a Model Reference Adaptive System, (source: Åstrom, 1995)

A block diagram of such system is shown in Fig7.1 The system has an ordinary feedback loop composed of the plant process and the controller and another feedback loop that changes the controller parameters. The controller parameters are changed on the basis of feedback from the error, defined as the difference between the output of the system and that of the reference model. The mechanism for adjusting the parameters in MRAS can be obtained in two ways, 1) application of stability theory, where the theory has been presented in previous sections, and 2) gradient methods, which we will discuss now.

### 7.2.2 MIT rule

The MIT rule was the original approach to MRAS control. It tells us how error is influenced by the adjustable control parameters. We consider a close loop system in which the controller has one adjustable parameter $\theta$. The desired closed loop response is specified by a model whose output is $y_m$. Let error

$$e = y - y_m$$

A possible way to adjust the parameter can be to reduce the loss function such as

$$J(\theta) = \frac{1}{2}e^2$$

is minimised. To make J small, it is reasonable to require the parameters to change in the direction of the negative gradient of J, that is,

**Equation 12**

$$\frac{d\theta}{dt} = -\gamma\frac{\partial J}{\partial\theta} = -\gamma e\frac{\partial e}{\partial\theta}$$

The partial derivative $\partial e/\partial\theta$ is called the *sensitivity derivative* of the system, and relates how the error is influenced by the adjustable parameter. If the parameter changes are slower than other variables in the system, then the derivative $\frac{\partial e}{\partial\theta}$ can be evalutated under the assumption that $\theta$ is constant.

There are of course other alternatives to the chosen loss function. For example it can be

$$J(\theta) = |e|$$

where its gradient method gives

**Equation 13**

$$\frac{d\theta}{dt} = -\gamma\frac{\partial e}{\partial\theta}sign(e)$$

We now present an example for 1st order system and then extend it to 2nd order systems for the problem we would like to solve.

**MRAS for first-order system**

Consider a system described by the model: [Åstrom and Wittenmark, 1995]

**Equation 14**

$$\frac{dy}{dt} = -ay + bu$$

Where u is the control variable and y is the measurement output. Assume that we want to obtain a closed-loop system described by

**Equation 15**

$$\frac{dy_m}{dt} = -a_m y_m + b_m u_c$$

Let the controller be given by

$$u(t) = \theta_1 u_c(t) - \theta_2 y(t)$$

The controller has two parameters. If they are chosen to be

$$\theta_1 = \theta_1^0 = \frac{b_m}{b}$$

$$\theta_2 = \theta_2^0 = \frac{a_m - a}{b}$$

The input-output relations of the system and the model are the same. This called perfect model-following. To apply the MIT rule, introduce the error

$$e = y - y_m$$

where $y$ denotes the output of the closed loop system. It follows from Eq. 14 and 15 that

$$y = \frac{b\theta_1}{p + a + b\theta_2} u_c$$

where $p = d/dt$ is the differential operator. The sensitivity derivatives are obtained by taking partial derivatives with respect to the controller parameters $\theta_1$ and $\theta_2$:

$$\frac{\partial e}{\partial \theta_1} = \frac{b}{p + a + b\theta_2} u_c$$

$$\frac{\partial e}{\partial \theta_2} = -\frac{b^2\theta_1}{(p + a + b\theta_2)^2} u_2$$

These formulas cannot be used directly because the process parameters $a$ and $b$ are not known. Approximations are therefore required. One possible approximation is based on the observation that $p + a + b\theta_2 = p + a_m$ when the parameters give perfect model-following. We will therefore use the approximation

$$p + a + b\theta_2 \approx p + a_m$$

Which will be reasonable when parameters are close to their correct values. With this approximation, we get the following parameters

$$\frac{d\theta_1}{dt} = -\gamma(\frac{a_m}{p + a_m})e$$

$$\frac{d\theta_2}{dt} = \gamma(\frac{a_m}{p + a_m})y$$

In these equations we have combined parameters $b$ and $a_m$ with the adaptation gain $\gamma'$, since they appear as the product $\gamma' b/a_m$. The sign of parameter $b$ must be known to have the correct sign of gamma. Note also that the filter has also been normalised so that its steady state gain is unity.

### 7.2.3 Self-Evolved MRAS controller

Indeed MRAS using MIT rules provide a convenient way of specifying the desired output and making the adaptive system conformed to that when controlling the plant process. In similar manner to genetic programming systems, we see the analogy, that we are actually making high-level statements to the adaptive system by stating the requirements (reduce the error with reference to the desired model) to the system.

Our goal is to have an entity that can automatically create a satisfactory program, given the goal and the requirements of the solution. It would not seem unreasonable then, that genetic programming might be able to evolve a controller (solution) to our above control problem. True to its nature, we now show that GP that uses evolutionary methods is capable of doing so. The purpose for our case is then to evolve a suitable controller that is able to base itself on the desired model to control the process, given the requirements (reduce the error) together with a suitable set of functions.

### 7.2.4 Set up

Since the original idea of MRAS is to tune the controller to give a satisfactory output of the plant given that the parameters of the plant process are unknown, we increase the difficulty for the GP system by loosening the requirement for MIT rule that the sign of the parameters must be known. Hence, what is given to the GP system is only the order of the process, without any prior knowledge of the sign or value of the parameters.

Furthermore, no hint was given about the MIT rule (assuming that the controller does not come from MIT). Hence there is no pre-determined strategy to how the controller should model the desired output. However, the necessary requirement is our high-level statements, namely to reduce the absolute error, which is to be input into our GP system. This of course, is conveyed through our fitness function.

Given the structure of RST controller is known, the end result of our solution should be one that gives the correct controller parameters $s_0, s_1, t_0, t_1, r_0$. In some sense, we can also view genetic programming in this case to automatically 'tune' the controller to the desired results. The block diagram of our experiment is set up as follows. (Figure 7.2)

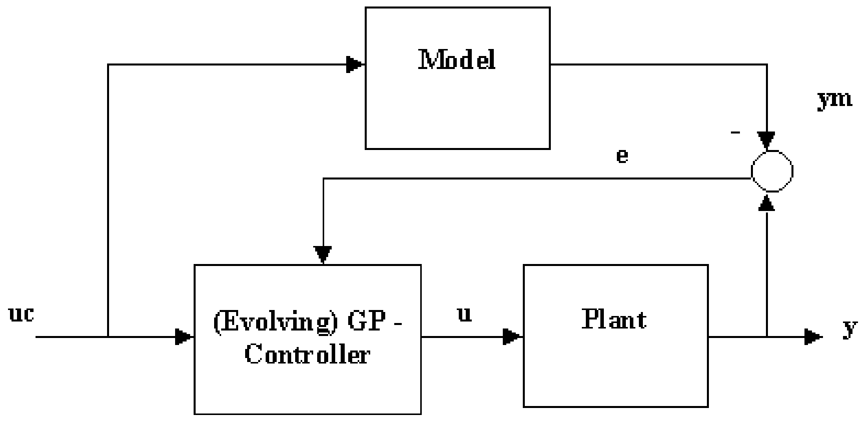


Figure 7.2: Block diagram of GP system design for MRAS controller

### 7.2.5 Functions Design and Individuals

We learnt from previous experiments that design of function sets for GP is crucial, since it would lend capabilities directly to the automatic creation of programs. The following we see how the functions can be designed.

A controller can be viewed to be made up of a summation of many small contribution of control signals, whether they are taken from feedback or feed forward loops. In this sense we can view our controller parameters or coefficients to consist of many small 'units' of coefficients also. For example, say the correct $s_0$ parameter for a controller is 2.5. We can interpret this as $2.5s_0 = (+1 + 1 + 0.5)s_0$. Or it can also be $(+1 + 0.5 + 0.5 + 0.5)s_0$ and the possibilities goes on. These $+1s_0$ and $0.5s_0$ can be interpreted as the functions for our individual program. A controller program in GP can hence to be seen in the form, for example,

$$+0.1s_0 + 0.1r_1 + 0.5s_0 + 0.5t_1 + 0.1r_1 + 0.5s_1$$

$$+0.1t_0 + 0.2t_0 + 0.1r_1 + 0.5t_1 + 0.1t_0 + 0.1t_1$$

and the controller will be

$$(s - 0.3r_1)u = (0.4t_0 + 1.1t_1)u_c + (0.6s_0 + 0.5s_1)y$$

Now the controller program can be easily subjected to genetic evolution using the genetic operations, since it is in a form similar to strings of code.

The function sets that are available for our GP systems are therefore of the form

$$s_0 = s_0 + 0.1$$

$$s_1 = s1 + 0.1$$

and so on. (see Section. 7.2.6)

The end result of our evolved program should be a controller that is able to give us the right coefficient to produce the desired model following.

Things would be easier if we know what are the signs of our controller parameters. However we assume no prior knowledge to that also. Therefore our functions sets have to be designed such that they include both the opposite parts of each other. It can be seen also, that the function set provided were very general and perhaps generous such that our GP system would have a free hand to select the required functions what it needed. Looking at the function sets alone there is no hint whatsoever anything that will help in our search. A random combination of these functions would most likely end up as a null controller anyway, giving zero coefficients.

### 7.2.6 Experiment

#### Initialisation of Individuals

Here we make a wild guess again, that the number of functions needed for a suitable solution to be 100 and we initialise a population of randomly generated individual programs each with a randomly selected string of functions containing within.

#### Fitness Criteria and Evaluation of Individuals

Evaluation on an evolved program was done by carrying that particular program through a simulation of test. The test is simply to see if the evolved controller is able to give an output from the process follow the desired output of two square waves. The evolved controller output $y$ is compared with the desired output $y_m$ and absolute error was used as criteria for fitness. However we note that reducing only the absolute error, is not enough, since the output can contain undesirable high peaks but with small error. Hence in our cost function we add an extra weight to $de/dt$, that is, minimise the difference of the error between 2 sample periods. That is to say, we award a controller which gives a stable output but higher absolute error than one that gives big overshoots but with lower absolute error.

**Parameters**

Fitness criteria: Reducing the absolute error between output and desired output from model. Also reduce undesirable output peaks.

Generations:100

Population: 80

Functions

$$s_0 = s_0 \pm 0.1, s_1 = s_1 \pm 0.1, s_0 = s_0 \pm 1, s_1 = s_1 \pm 1$$

$$t_0 = t_0 \pm 0.1, t_1 = t_1 \pm 0.1, t_0 = t_0 \pm 1, t_1 = t_1 \pm 1$$

$$r_1 = r_1 \pm 0.1, r_1 = r_1 \pm 0.01, \text{and a function that returns null}$$

Notice some functions are $\pm 0.1$ and some are $\pm 1$, these are to provide cases where the coefficients need to be large, yet may sometimes need some acurracy. This is the case for $r_1$ where we want the feedback control signal to be small yet exact. A null function was also provided for the case where no increase or decrease in the coefficents is needed.

### 7.2.7 Results

The following gives an account of the results of one particular experiment. However it should be said that although GP would be able to give similar results to a single problem most of the time, it is rare however, that the path taken by it is the same for all generations. That is, it is difficult to predict the route which evolution chooses to take. Some runs would be better than the rest while some may not. However, these results do speak of *in general* how a typical run of GP on the self-evolved MRAS controller looks like.

The process which we would like to control is

$$\frac{k}{s^2 + a}$$

where $k$ and $a$ are unknown process parameters.

It is needed to convert this to discrete time, they are calcalated as of the form:

$$y(t) = 1.96y(t-1) - y(t-2) + 0.002791u(t-1) + 0.002791u(t-2)$$

and the process model is

$$y_m(t) = 0.0182u_c(t-1) + 0.01657u_c(t-2) + 1.721y_m(t-1) - 0.7558y_m(t-2)$$

Naturally it should not be expected that GP can give correct answer immediately just after the first few generations. Although it is expected that when averaged out, the population of programs would produce a null controller that would give no output, there exists nonetheless, some individuals that performs better than the rest. Sometimes in this initial stage, the best controller from its generation could be one which is selected as long as there was some output control signal. Sometimes it could even be an unstable controller that blew up the process. This should not come as a surprise since human beings started as ape-like creatures too, for people who believe in evolution anyway. Figure 7.3 and Figure 7.4 shows the output of the process and control signal of the best of 1st generation.

For the experiments, unknown process parameters are taken as $k = 0.14$ and $a = 1$

$$k = 0.14, a = 1$$

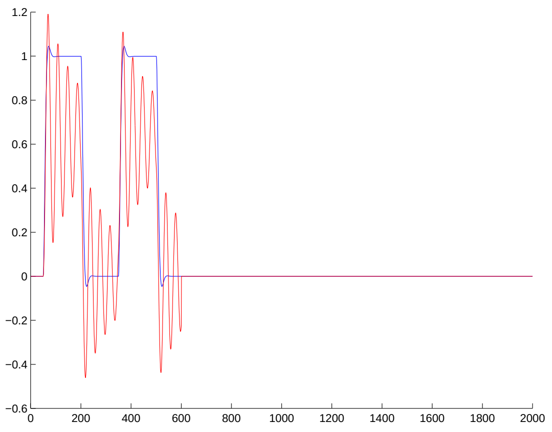


Figure 7.3: output signal of best solution against model reference: generation 1

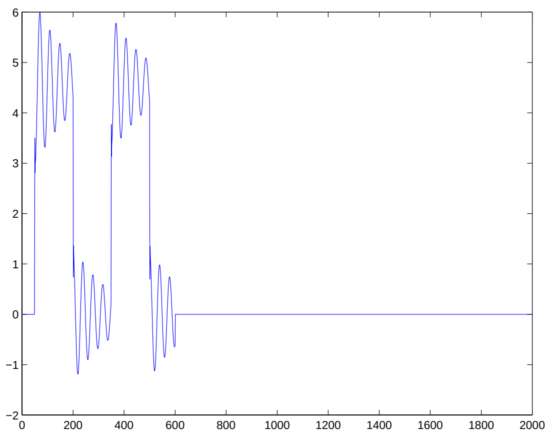


Figure 7.4: control signal of best solution: generation 1

Without losing generality and variation, typically by 10 generations a population of around 80-100 was able to produce the best individual of fitness around

0.15 (with peaks and bad response taken into account), that is, its absolute error compared to the model is around 6.7. (Figure 7.5) Also, by looking at the coefficients of the evolved best-of-generation individual controllers, it can be seen that the system does not go after a greedy 'hill-climbing' approach. This is apparent since the coefficients might change their own sign, or vary in opposite directions in terms of magnitude.

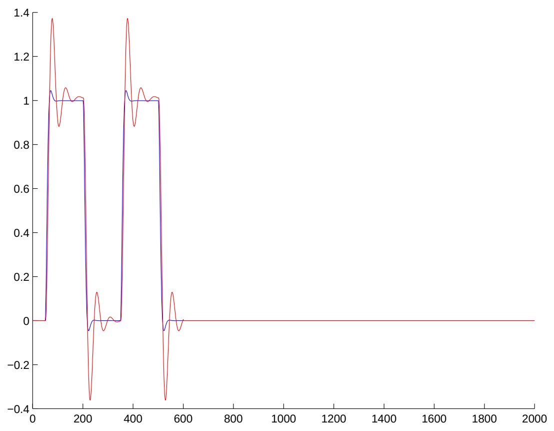


Figure 7.5: output signal of best solution against model reference: generation 11

By around 45 generations and above, some rough prelimary results was produced. The controller output is shown in Figure 7.6.

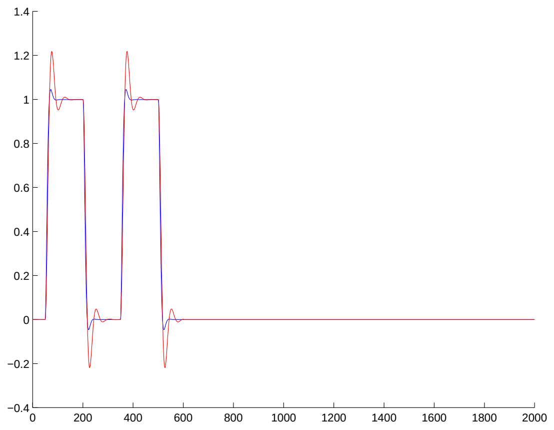


Figure 7.6: output signal of best solution: generation 45

During implementation of the experiment, GP was required to follow the two square waves produced by the model reference output. However, during the later part, only one square wave was used to save computational time of the fitness trials. Such variations in the test trials in experiments should not, and did not affect the outcome of the solution. This is because since GP depends on fitness functions to drive a population towards the required goal, and fitness function is a *comparative* contest between individuals, the results remain unaffected. In our case the fitness function is to reduce the absolute errors and also the output peaks, although the values of the fitness points might not be the same as previous experiments using two square waves, the best individual program is still chosen.

It could be seen that by generations around 70 and above, an almost perfect model following was achieved. The coefficients of the controller given by the GP systems are

$$t_0 = 1.8000$$

$$t_1 = 6.2000$$

$$s_0 = -45.8000$$

$$s_1 = 45.3999$$

$$r_1 = -0.0400$$

and hence the controller is of the form

$$(s - 0.04)U(s) = (1.8s + 6.2)U_c(s) - (-45.8s + 45.4)Y(s)$$

The output of the process and control signal of two of the best individual programs from the GP run are shown in Figure 7.7 and 7.8 respectively.

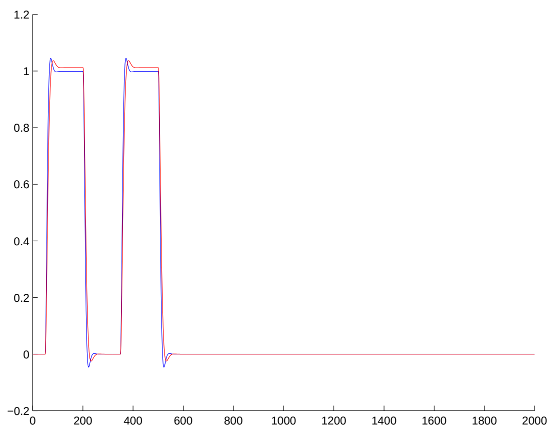


Figure 7.7: output signal of best solution: generation 72

Of course, there are also ocassions where even better results were produced. For example see Figure 7.11, where the size of population is 100 instead.

The program of the controller that generated the coefficients for the controller typically look like this,

```
Columns 1 through 12
15 15 11 15 13 15 15 7 15 13 7 15
Columns 13 through 24
15 11 15 15 15 15 13 15 15 15 13 14
```

where each number represents the index of the functions used.

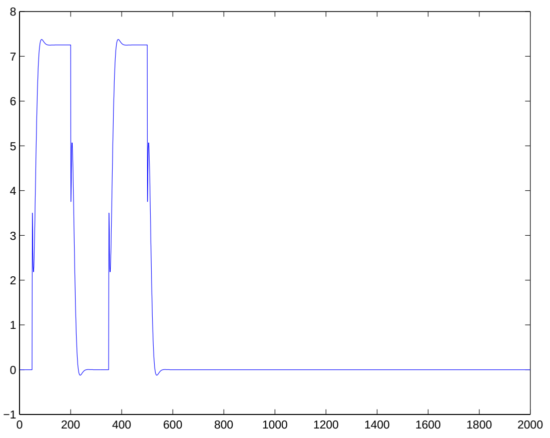


Figure 7.8: control signal of best solution: generation 72

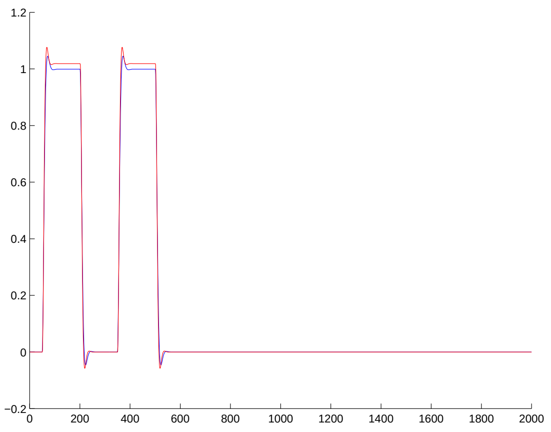


Figure 7.9: output signal of best solution: generation 78, result of the GP run

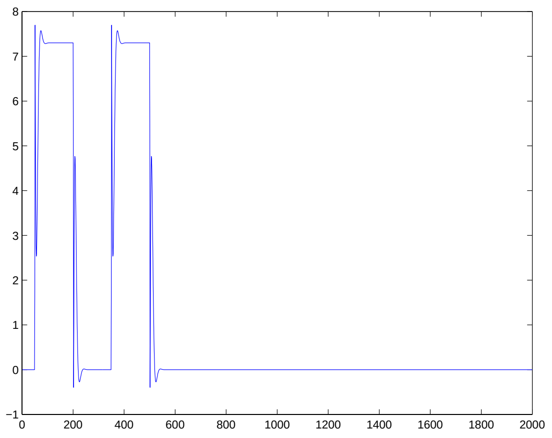


Figure 7.10: control signal of best solution: generation 78, result of the GP run

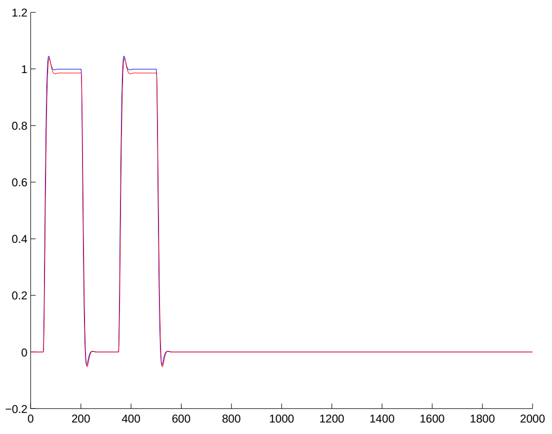


Figure 7.11: output signal of best solution: generation 90

### Change in Process Parameters

Adaptivity to changes in the process environment is a sought after property of MRAS controllers. A change in the process parameters was introduced to see the effect of the generation of solutions, after model following had been achieved. In our case, the parameters $k$ and $a$ were increased to three times its orginal value, after initial perfect model following has been produced.

As again, we would not expect GP to change rapidly towards the right solution, since no prior iniformation about the change was given, or estimation of time-varying parameters carried out. Perhaps even if these were given, we would still expect GP to automatically create a solution without being given explicitly the mechanisms for such changes, i.e. if we know that the process changes in certain direction, we could probably inject a biased probability of some functions to appear against the rest of the functions. This is not done, however since we let GP self-discover the changes and tune towards the correct solution automatically without help instead. It was seen that within a span of 50 generations, the correct controller solution could be produced to give perfect model following. Figure 7.12 to Figure 7.14 traces the best-of-generations when the process parameters $k$ from 0.14 to 0.42,$a = 1$ to $a = 3$.

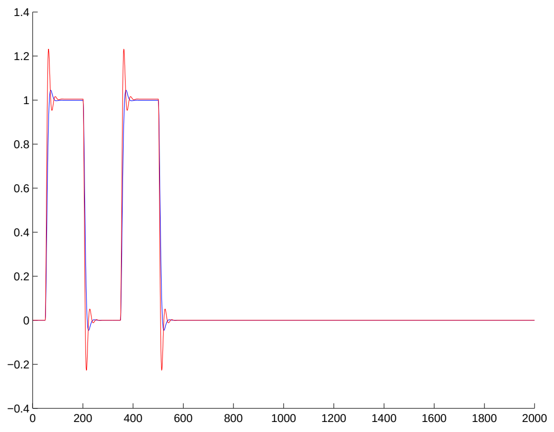


Figure 7.12: output signal after process parameters were changed: generation 1

It can be seen from Figure 7.12 that within the population of programs there may exist some individuals which may be able to give some initial rough results. By around 5 generations and above better results were produced (Figure 7.13) and by generation 20 and above perfect modelling could be produced. (Figure 7.14)

The resulting controller coefficients are:

$$t_0 = 2.5000$$

$$t_1 = 0.5000$$

$$s_0 = -12.000$$

$$s_1 = 17.000$$

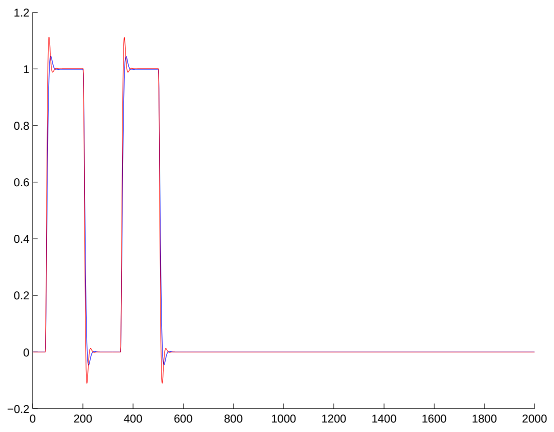


Figure 7.13: output signal after process parameters were changed: generation 5

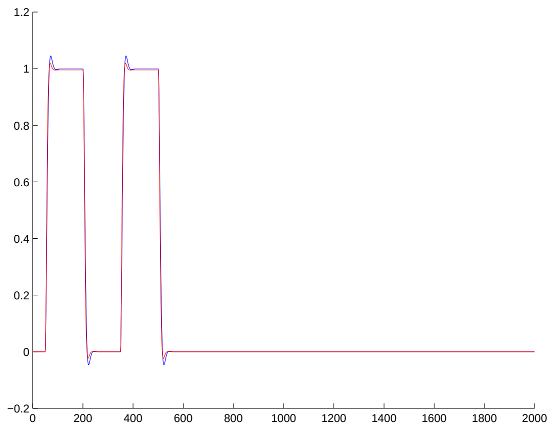


Figure 7.14: output signal after process parameters were changed: generation 22

$$r_1 = -0.1200$$

Here we turned ambitious and go to further extremity, the process was then changed sign and observed after the original perfect model following of the old process was achieved. As expected the controller first swung to instability, but after a span of more than 100 generations, the best indivdual was able to give response in the correct direction as the model, although the results was rough and it never fully recovered. (Figure 7.15)

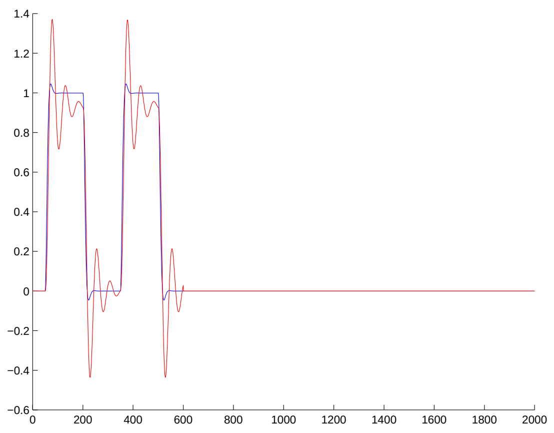


Figure 7.15: output signal after process parameter was changed to $-k$: generation 22

### Signs of Process Unknown

The sign of the process was assumed to be unknown, hence in this case we could put it to negative of the original, while maintaining the same model reference. As like in section [7.2.7] in earlier part, the correct controller could be found. Of course in this case the control signal is of the opposite sign.

Process with negative $k$ was tried out in this case with other parameters remaining the same. Output gives similar results but with the control signal being negative. See Fig 7.16 and Fig 7.17.

### Mutation Operations

Mutation operation was used in this experiment because it is believed that it should be able to help the GP run since it gives the population a more varied and diverse selection. Also, some functions/genes which died away but might be useful in small number might be introduced back into the population. This may be somewhat contrary to the Price's Theorem stating that healthy genes propagate in the evolution. However in our case the process environment is not static, and a goal of adaptivity may justify the use of the operation, since probably the mutated individual is the lucky one that may bring the evolved solution closer to the required one. Mutation operations stand at around 10% among all the genetic operations in this experiment.

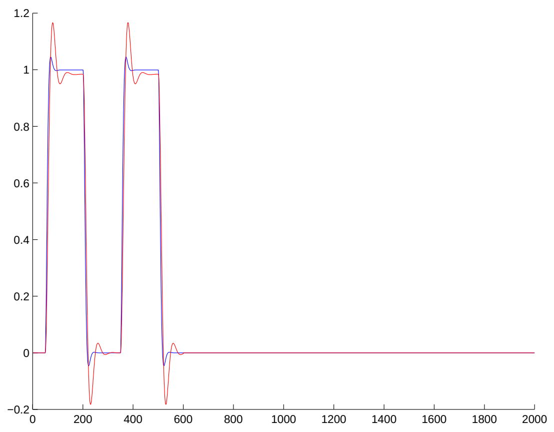


Figure 7.16: output signal with process $-k$: generation 61

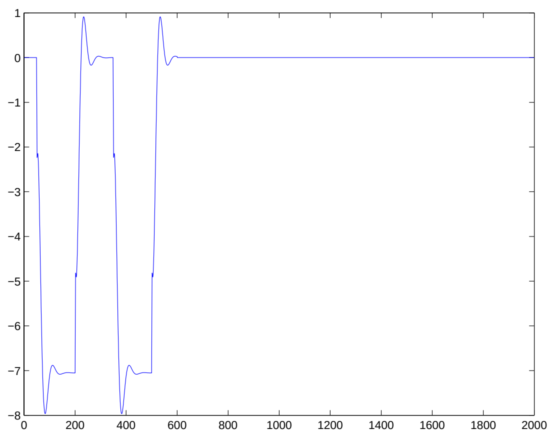


Figure 7.17: control signal with process $-k$: generation 61

### 7.2.8 Search Space of the Self-Evolved MRAS Controller

Using similar arguments as that of section 7.1.5, the search space in this case can be roughly calculated as follows. As again, we make an initial guess of the required functions for an individual, usually with lower limit, say 100. Number of functions in the experiment is 23, and there is no need for correct sequence and arrangements for the functions in each program, however we do not know the exact combination of the correct coefficient. Hence we get

Initial no. of functions in Individual: $I = 100$
No. of functions available: $f = 23$
total search space is: $f^I = 23^{100}$ = some big number

### 7.2.9 Some Properties of the Self-Evolved MRAS Controller

As can compared with section [7.2.2] where calculation using the MIT rule was used, we see that our genetic programming system does nor know any of that rule or method. Further, whereas the sign of the process is assumed to be known in MIT rule method, GP in is case to not assume this prior knowledge. In fact, GP does not explicitly do calculations for the required controller. The only clue that it received from the human designer is the required high-level statements conveyed through the designed fitness function. Also, no explicit estimation of process parameters were carried out for system identification.

The only main tools which guided GP in the search were the fitness functions and the mechanisms of genetic operations. Trial and selections which individuals within the population would survive played the role of designing. Since there was no assumption on the sign of the process parameters, GP showed a little more robustness than the MIT-rule designed MRAS. Also, it showed some amount of adaptivity to process variations if the change is within suitable limits. However, the main drawbaack of GP is the computational time required for the reaching the solution. Doing genetic operations on individuals are relatively fast, but not the trials and tests which is used to measure the fitness of each individual programs.

#### Rate of Convergence and Global Optima

It should be expected that the fitness functions, number of generation and size of population would affect the rate of convergence and optimality of solutions. Furthermore, probably the choices of the probability of various genetic operations may have some effect too. Unfortunately, these are not known for sure to have any influence, or if so, to what extent. As far as the experiment went, the choice of population size between 50 to 200 did not produce significant changes in rate of convergence, or affect the quality of the solutions. Perhaps there might be some relationships, but further works need to be done to justify either of the claim.

Rather, it does appear that the selection process of individuals from the population may affect the rate of convergence and also optimality, particularly when the tournament selection technique was employed. (see Appendix on Matlab Code). There is general belief that the larger the number of individuals selected to participate in the tournament, the faster is the rate of convergence.

However global optimal might not be reached as GP converges to local optima, since variations within the genepool might be lost. As again, more works need to be done in this area.

### Scalability of the Problem

This is left to be explored too, regarding the scalability of GP to tackle problems of higher order. There are on going hot debates within the GP community in this area. For now, it is suggested that as a continuation to this thesis, the self-evolved MRAS could be extended to problems of higher order to see if GP is still possible to automatically generate solutions to them. This could be automatic choice of the order of controller, including if necessary, any observer involved.

## 7.3 Conclusion

In the last two experiments, we have shown that genetic programming was able to solve automatically some non-trival control problems. The results obtained from these self-evolved programs were satisfactory, and in some cases of GP run, equally good compared to solutions produced by the human designer. Most of the time, rough results could be guaranteed.

In the case of discrete time Lyapunov solver, it has been shown that GP is cabable of solving problems requiring some form of iterations or recursion, memory and correct sequences of selected functions. For the self-evolved MRAS controller, we see that GP is able to without any prior knowledge of process parameters, generate a controller based on model following. It showed some form of adaptivity in response to changes in process parameters. Besides that, its non-greedy approach, instead of solely hill-climbing to the problem, was also demonstrated.

# Chapter 8

# Conclusion

This thesis has demonstrated some applications of genetic programming in solving control problems. The mechanisms used in this thesis if the reader has not already noticed, is remarkedly easy. Furthermore, it does not based itself on strict mathematical analysis, but rather, the qualitative nature of each solution. It solves all the problems using the basic mechanisms of fitness functions and genetic operations. Hence the choice of fitness functions and the designed functions for GP run is particularly important.

In most cases GP was able to produce at least some rough solutions as long as its fitness criteria is reasonably suitable. Hence it seems powerful in the sense that so long that problem representation is done correctly, the method of genetic programming is able to solve a wide variety of problems, with only these few simple genetic tools. One might then be tempted to assume that it is generally a good method to solve all problems. However, as NFL theorem states, there is no single best universal method to problems, therefore it is highly recommended that GP is used with some caution.

Nonetheless, although there is no best way to things, there are still methods that are difficult or easy to implement. Perhaps genetic programming represents the latter, hopefully shown throughout the thesis. Most importantly, genetic programming showed the feature of using what is prior-known and what is unknown to its advantage. However, intuition of the human designer to the approach of each problem is still indispensable, perhaps even more important, for genetic programming to work. Therefore it is probably to a large extent, the combination of the intuition of the human mind with the computational power of the computer using evolutionary approach that would make GP able to create programs and solutions automatically. In this thesis it is shown that genetic programming can be, and have been applied to the field of control engineering in a variety and manner of problems.

# Bibliography

[1] Altenberg, L.(1995). The Schema Theorem and Price's Theorem. In Whitley, L.D. and Vose, M.D., editors, *Foundations of Genetic Algorithms 3*, pages 23-49, Estes Park, Colarado, USA. Morgan Kaufmann.

[2] Andre, D. (1994). Automatically defined features: The simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them. In Kinnear, Jr., K.E. editor, Advances in Genetic Programming, chapter 23. MIT Press.

[3] Alexander, I. (1997). *Impossible Minds: My Neurons, My Consciousness.* Imperail College Press.

[4] Åstrom, K.J., Wittermark, B. (1995). *Adaptive Control*, 2nd Edition, Addison-Wesley Publishing.

[5] Åstrom, K.J., Wittermark, B. (1997). *Computer Control Systems: Theory and Design.* Prentice Hall Information and System Sciences Series.

[6] Boyd, S., Ghaoui, L.E., Feron, E., Balakrishnan, V., *Linear Matrix Inequalities in System and Control Theory*, 1994. SIAM.

[7] Chellapilla, K.,(1997). Evolutionary programming with tree mutations: Evolving computer programs without crossover. In Koza, J.R., Ded, K., Dorigo, M.. Fogel, D.B., Garzon, M., Iba, H., and Riolo, R.L., editors. *Genetic Programming 1997: Preceedings of the 2nd Annual Conference*, Standford University, CA, USA. Morgan Kaufmann.

[8] Chipperfield,A.J., Fonseca,C.M and Fleming, P.J..Development of genetic optimization tools for multi-objective optimization problems in CACSD. In *IEE Colloquium on Genetic Algorithms for Control Systems Engineering.* Digest 1992/106.

[9] Chu, P.C., Beasley, J.E. (1995). *A Genetic Algorithm for the Set Partitionning Problem.* The Management School, Imperial College, London, April 1995

[10] Davis, L. editor (1991). Schedule Optimisation Using Genetic Algorithm.*Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York.

[11] Davis, L., Ed. *Genetic Algorithms and Simulated Annealing.* Morgan Kaufmann Publishers, 1987.

[12] Glad, T., Ljung, L., *Reglerteori,Flervariable och olinjara metoder.* Studentlitteratur.

[13] Goldberg, D.E. (1989).Genetic ALgorithms in Search Optimization and Machine Learning. Addison-Wesley.

[14] Goldberg, D.E. (1992). Massive Modality, Deception and genetic algorithms. In Manner, R. and Manderick, B. , editors. *Parallel Problem Solving from Nature 2*, Brussels, Belgium. Elsevier Science.

[15] Holland, J.H. (1973). Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computation,* 2:88-105.

[16] ] Hunt, K.J., (1992)Ploynomial LQG and H1 controller synthesis: A genetic algorithm solution. In *Proceedings of the IEEE Conference on Decision and Control*, Tucson, USA.

[17] Johansson, M., Bernhardsson, B., Johansson, K.H., Exercise in Nonlinear Control Systems, 2000. Department of Automatic Control, LTH.

[18] Johansson, R. (1993). System Modeling and Identification. Prentice Hall.

[19] H.K. Khalil, *Nonlinear Systems*, 2nd Edition. Prentice-Hall, 1996.

[20] Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT Press, Cambridge Massachusetts.

[21] Koza, J. (1995). *Genetic Programming II: Automactic Discovery of Reusable Programs.* MIT PRess. Cambridge Massachusetts.

[22] Koza, J., Bennett III, F.H., Andre,D. (1996). Four Problems for which a computer program evolved by genetic programming is competitive with human performance. In *Preceedings of the 1996 IEEE International Conference on Evolutionary Computation,* Vol1., IEE Press.

[23] Koza4) Koza, J., Bennet III, F.H., Andre, D., Keane, M.A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving.*Morgan Kaufmann.

[24] Kinnear, Jr., K.E. 1994 editor. *Advances in Genetic Programming*, Chapter 1. MIT Press.

[25] Langdon, W.B. (1998) *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, Kluwer Academic Publishers.

[26] Loan, C.F.V. (1997), *Introduction to Scientific Computing, A Matrix-Vector Approach Using Matlab*, Matlab Currriculum Series.

[27] Nolfi,S., Elman, J.L. and Parisi, D.. *Learning and evolution in neural networks.* Institute of Psychology, C.N.R.-Rome. Technical Report 94-08. 1994.

[28] Samueal, A. (1959). Some studies in machinge learning using the game of checkers. *IBM Journal of Research and Development.*

[29] Slotine, J.J., Li, W. (1991). *Nonlinear Control.* Prentice Hall.

[30] Tackett, W.A. (1993). Genetic Programming for feature discovery and image discrimination. In Forrest, S.,editor. *Preceedings of the 5th International Conference on Genetic Algorithms, ICGA-93.* University of Illinois at Urbana-Champaign. Morgan Kaufmann.

[31] Wolpert, D.H, Macready, W.G. (1996). No free Lunch theorems for search. SFI-TR-95-02-010.

[32] Wolpert, D.H. and Macready, W.G. (1997). No free lunch theorems for optimization. *IEEE Transaction on Evolutionary Computation.*

# Appendix A

# Matlab Code for Discrete Lyapunov solver

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Function GPmain is the main program that calls all other programs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [best] = gpmain()
%function returns the best individual program from the GP run,
%and also display some results

global Functions Geneoperations MaxDepth %genepoolfitness genepoolresults
global Stopflag matrow matcol population ADFrecord ADFDepth
global VarConst numvariables numconstants Stopflag
global Individual IndividualDepth IndividualFitness best

matrow = 2;
matcol = 2;
population = 70;                        %define population size
maxstring = 150;              %define the maximum length of the string to
%describe the operations of the genome
maxdepth = 50;                 %define the maximum depth of the genome
numgenerations = 2;            %define the number of generations for the run
matrixspace = 500;             %define the amount of matrix space to allocate
%will be used as memory for variables
Stopflag = 0;                  %Stopflag, global flag for termination criteria
bestfit = 0;                   %define initial best fitness

numvariables = 2;              %define number of variables the run going to use
numconstants = 2               %define number of constants here
numADFs = 100;                 %define number of initial ADFs here
ADFDepth = 20;                 %define the depth of each function
IndividualDepth = 300;     %definfe the depth and complexity of each individual
noplayers = 10;                 %define number of players in tournament game

results = []; varindex = []; espression = []; tmpfit = zeros(population,1);
```

```
format long;

%Define functions and genetic operations
%enter function names here, but name must be 5 letters
%Functions = ['addit';'subtr';'atpaf'];

Functions = ['addi1';'addi2';'addi3';'addi4';'subt1';'subt2';'subt3';'subt4'; ...
      'atv11';'atv12';'atv21';'atv22';'pav11';'pav12';'pav21';'pav22'];


%enter gene operations here, but name must be 5 letters
Geneoperations = ['cross'; 'mutat'; 'repro'];   %name must be 5 letters

%initialise initial population
initvariables(numvariables, numconstants);

initIndividual(population)

%evaluate initial fitness of the defined functions
VarConst

%evaluate initial fitness of individuals
for pop = 1:population
   if Stopflag == 0
      initvariables(numvariables, numconstants);
      tmpfit(pop) = evalfitind(Individual(pop,:));
   end
end
%Individual
tmpfit
tmpmax = 0;
for gen = 1:25
   if Stopflag == 0
      tmpNewGen=zeros(population,IndividualDepth);
      tmpfit = zeros(population,1);

      NewPop = 1;
      while NewPop <= population

         chooseop = ceil(4*rand(1));
         if chooseop ==1                                 %25% reproduction
            NewInd = repro(noplayers);
            tmpNewGen(NewPop,:)=NewInd;
            tmpfit(NewPop) = evalfitind(NewInd);

            NewPop = NewPop+1;
         else                                            %75% crossover

            [NewInd1, NewInd2] = cross(noplayers);
            tmpNewGen(NewPop,:)=NewInd1;
            tmpfit(NewPop) = evalfitind(NewInd1);

            NewPop = NewPop+1;
            if NewPop <= population
```

```
               tmpNewGen(NewPop,:)=NewInd2;
               tmpfit(NewPop) = evalfitind(NewInd2);
            else
            end

         end

      end
      %tmpNewGen
      IndividualFitness = tmpfit;
      Individual = tmpNewGen;
      [mm,imax]=max(IndividualFitness);

      if mm > tmpmax      %compare the best of generations with record
         bestind = Individual(imax,1:IndividualDepth);
         initvariables(numvariables, numconstants);

         %[finalVAR,iniVAR, espression]=evalindividual(bestind);
         [finalVAR,espression]=evalindividual(bestind);
         finalVAR
         %showfit = evalfitind(bestind)
         tmpmax = mm
         genrec = gen
      else
      end
      disp('one generation');
      % tmpfit
      % meanfit = mean(tmpfit);
      % [finalVAR,iniVAR, espression]=evalindividual(Individual(1,:))
   end
end

%Individual
%display some evaluation of results
disp('1 0; 0 2');
initvariables(numvariables, numconstants);
[finalVAR,espression]=evalindividual(bestind)

disp('2 0 ; 0 2');
initvariables2(numvariables, numconstants);
[finalVAR,espression]=evalindividual(bestind)

disp('1.4 0.5 ; 0.5 1.65');
initvariables3(numvariables, numconstants);
[finalVAR,espression]=evalindividual(bestind)

disp('')
initvariables3(numvariables, numconstants);
[finalVAR,espression]=evalindividual(bestind)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Genetic Operations: Reproduction and Crossover
%Mutation operation is not done.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function NewInd = repro(noplayers)
%reproduces the same genome for the next generation
global Individual

index = selectindividual(noplayers);

NewInd = Individual(index,:);

function [NewInd1, NewInd2] = cross(noplayers)
%choose a parent according to the probability of the fitness of the genepool
%do cross over and return the New Individual

global Individual

sizeind = size(Individual,2);
NewInd1 = zeros(1,sizeind);
NewInd2 = zeros(1,sizeind);
index1 = 1; index2 = 1;

while index1==index2
   index1 = selectindividual(noplayers);
   index2 = selectindividual(noplayers);
end

parent1 = Individual(index1,:);
parent2 = Individual(index2,:);

[tmpfragment1, A1,A2] = getfrag(parent1);
[tmpfragment2, B1,B2] = getfrag(parent2);

[Row,P1end] = find(parent1 == 9999);            %get the last node
NewInd1frag = [parent1(1,(1:A1-1)), tmpfragment2, parent1(1, A2+1:P1end)];
[Row,P2end] = find(parent2 == 9999);            %get the last node
NewInd2frag = [parent2(1,(1:B1-1)), tmpfragment1, parent2(1, B2+1:P2end)];

NewInd1(1:size(NewInd1frag,2))= NewInd1frag;
NewInd2(1:size(NewInd2frag,2))= NewInd2frag;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Evaluation of Individuals
%Calcalate results and fitness of individuals.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [finalVAR,expression]=evalindividual(individual)
%evaluate the fitness of one individual
%model tests with various order are used for judgement

global VarConst Functions numvariables
results =[]; iniresult=[];

[rr,endnode] = find(individual==9999);
expression = [''];
```

```
for eachfun = 1:endnode-1
   functn = individual(1,eachfun) ;
   expression = feval(Functions(functn,:),expression)   ;
end

finalVAR = VarConst(1:numvariables,:);
%espressionA = espression;

function fitness = evalfitind(individual)
%return the fitness of one individual

global Stopflag VarConst matrow matcol numvariables numconstants best

espressions = [];

initvariables(numvariables, numconstants);        %3 iterations
[finalVAR,espressions]=evalindividual(individual);
Pk = store2mat(VarConst(1,:), matrow, matcol);
Pk_1 = store2mat(VarConst(2,:), matrow, matcol);
A = store2mat(VarConst(3,:), matrow, matcol);
Q = store2mat(VarConst(4,:), matrow, matcol);

fitmat1 = A'*Pk*A - Pk + Q;
fit1 = sum(sum(abs(fitmat1)));

initvariables2(numvariables, numconstants);      %2 iterations
[finalVAR,espressions]=evalindividual(individual);
Pk = store2mat(VarConst(1,:), matrow, matcol);
Pk_1 = store2mat(VarConst(2,:), matrow, matcol);
A = store2mat(VarConst(3,:), matrow, matcol);
Q = store2mat(VarConst(4,:), matrow, matcol);

fitmat2 = A'*Pk*A - Pk + Q;
fit2 = sum(sum(abs(fitmat2)));


initvariables3(numvariables, numconstants);    %5 iterations
[finalVAR,espressions]=evalindividual(individual);
Pk = store2mat(VarConst(1,:), matrow, matcol);
Pk_1 = store2mat(VarConst(2,:), matrow, matcol);
A = store2mat(VarConst(3,:), matrow, matcol);
Q = store2mat(VarConst(4,:), matrow, matcol);

fitmat3 = A'*Pk*A - Pk + Q;
fit3 = sum(sum(abs(fitmat3)));


initvariables4(numvariables, numconstants);    %4 iterations
[finalVAR,espressions]=evalindividual(individual);
Pk = store2mat(VarConst(1,:), matrow, matcol);
Pk_1 = store2mat(VarConst(2,:), matrow, matcol);
A = store2mat(VarConst(3,:), matrow, matcol);
Q = store2mat(VarConst(4,:), matrow, matcol);
```

```
fitmat4 = A'*Pk*A - Pk + Q;
fit4 = sum(sum(abs(fitmat4)));

initvariables5(numvariables, numconstants);      %6 iterations
[finalVAR,espressions]=evalindividual(individual);
Pk = store2mat(VarConst(1,:), matrow, matcol);
Pk_1 = store2mat(VarConst(2,:), matrow, matcol);
A = store2mat(VarConst(3,:), matrow, matcol);
Q = store2mat(VarConst(4,:), matrow, matcol);

fitmat5 = A'*Pk*A - Pk + Q;
fit5 = sum(sum(abs(fitmat4)));

initvariables6(numvariables, numconstants);    %8iterations
[finalVAR,espressions]=evalindividual(individual);
Pk = store2mat(VarConst(1,:), matrow, matcol);
Pk_1 = store2mat(VarConst(2,:), matrow, matcol);
A = store2mat(VarConst(3,:), matrow, matcol);
Q = store2mat(VarConst(4,:), matrow, matcol);

fitmat6 = A'*Pk*A - Pk + Q;
fit6 = sum(sum(abs(fitmat4)));

initvariables7(numvariables, numconstants);      %10 iterations
[finalVAR,espressions]=evalindividual(individual);
Pk = store2mat(VarConst(1,:), matrow, matcol);
Pk_1 = store2mat(VarConst(2,:), matrow, matcol);
A = store2mat(VarConst(3,:), matrow, matcol);
Q = store2mat(VarConst(4,:), matrow, matcol);

fitmat7 = A'*Pk*A - Pk + Q;
fit7 = sum(sum(abs(fitmat2)));


%fit = fit1 + fit2 + fit3 + fit4 + fit5 + fit6;
fit = fit1 + fit2 + + fit3 + fit4 + fit5 + fit6 +fit7;

if fit <= 0.05
   disp('fitness met for individuals')
   Stopflag = 1;
   [finalVAR, espression]=evalindividual(individual)
   fitness = fit;
   best = individual
else
   fitness = 1/fit;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Minor Functions for Storage, Retrieval, and Indexing
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [Tmpgenestring, nA, nB]= getfrag(genomeA)
```

```
%for crossover, the function return a fragment of the genome
%to be used for the crossover
%operation.

nodeA1 = 1;
nodeA2 = 1;

nodeA1 = pickanode(genomeA);
nodeA2 = pickanode(genomeA);

if nodeA1 == nodeA2
   nA = nodeA1;
   nB = nodeA2;
   Tmpgenestring = genomeA(nodeA1);
else

   nodeA = [nodeA1, nodeA2];
   nodeA = sort(nodeA);   %do a sort so that node A1 < node A2
   nA = nodeA(1,1);
   nB = nodeA(1,2);
   Tmpgenestring = genomeA(:,nA:nB);    %the fragment is [func,B,C,...,C]
end

function genomerow = convert4store(genemat)
%convert a (r x c) matrix to a row vector (of size [1,r x c])to be used for
%storage of genepool. see also 'store2mat'.
genomerow = [];
[row,col]=size(genemat) ;
for r = 1:row
   for c = 1:col
      genomerow = [genomerow genemat(r,c)];
   end
end

function genemat = store2mat(genomerow,row,col)
%convert a row vector (of size [1,r x c]) genomerow to a (r x c) matrix
%genemat to be used for evaluation. see also 'convert4store'.

s = 1;
genecolsize = size(genomerow,2);
givensize = row*col;
if givensize == genecolsize
   for r = 1:row
      for c = 1:col
         genemat(r,c) = genomerow(s);
         s = s + 1;
      end
   end
else
   disp('row and col size does not match dimension of input');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Selection of individaul programs for genetic operations
```

```
%via Tournament method
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function index = selectindividual(noplayers)
%select individuals from genepool via tournament methods
%returns the index of the individual from Individualfitness

global IndividualFitness population

indfit = zeros(noplayers,1); bestfit = 0;

tournamentpool = ceil(population*rand(noplayers,1));  %index of the players

for player = 1:noplayers
   indfit(player)= IndividualFitness(tournamentpool(player,1));
   if bestfit <= indfit(player)
      bestfit = indfit(player);
      index = tournamentpool(player,1);
   else
      index = tournamentpool(1,1);

   end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Initialisation of Individuals and Variables
%only one example of initialisation of variables shown
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function initIndividual(pop)
%function initialise a number of individuals
global Individual IndividualDepth IndividualFitness Functions best

sizeofFunction = size(Functions,1);
%sizeofADFrec = size(ADFrecord,1);
Individual = zeros(pop,IndividualDepth);

for i=1:45
%Individual(1:pop,i) = ceil(rand(pop,1)*(sizeofADFrec));
Individual(1:pop,i) = ceil(rand(pop,1)*(sizeofFunction));

end
Individual(1:pop,i+1) = 9999;
%Individual
IndividualFitness = zeros(pop,1);

best = zeros(1,IndividualDepth);


function initvariables(novariables, noconstants)
%initialise number of variables and constants here
global matcol matrow VarConst

Variables = ones(novariables,matrow*matcol);
```

```
Constants = ones(noconstants,matrow*matcol);
VarConst = [Variables;Constants];


%T = 0.9*rand(2,2);
%T =[ 0.43365 0.44736; 0.27610 0.07882;];
%T = [0.21448 0.87222; 0.73935 0.76453];

A =[ 0.06858167   0.0062523721; 0.28649129   0.28058822];  %3 iterations
   Q = [1 0; 0 1];

VarConst(3,:) = convert4store(A);
VarConst(4,:) = convert4store(Q);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Functions which can be used by the Genetic Programming System
%to Evolve its solution.
%Boring but necessary
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function expr = addi1(expr)
%Var1 = Var1 + Q

global VarConst

P1 = VarConst(1,:);
Q = VarConst(4,:);

VarConst(1,:) = P1 + Q;
expr = [expr,'v1=v1+Q;'];

function expr = addi2(expr)
%Var2 = Var1 + Q

global VarConst

P1 = VarConst(1,:);
Q = VarConst(4,:);

VarConst(2,:) = P1 + Q;
expr = [expr,'v2=v1+Q;'];

function expr = addi3(expr)
%Var1 = Var2 + Q

global VarConst

P1 = VarConst(2,:);
Q = VarConst(4,:);

VarConst(1,:) = P1 + Q;
expr = [expr,'v1=v2+Q;'];

function expr = addi4(expr)
%Var2 = Var2 + Q
```

```
global VarConst

P1 = VarConst(2,:);
Q = VarConst(4,:);

VarConst(2,:) = P1 + Q;
expr = [expr,'v2=v2+Q;'];

function expr = atv11(expr)
%Var1 = A'Var1

global VarConst

A1 = VarConst(3,:);
A = store2mat(A1,2,2);

P1 = VarConst(1,:);
P = store2mat(P1,2,2);

VarConst(1,:) = convert4store(A'*P);
expr = [expr,'v1=At*v1;'];

function expr = atv12(expr)
%Var2 = A'Var1

global VarConst

A1 = VarConst(3,:);
A = store2mat(A1,2,2);

P1 = VarConst(1,:);
P = store2mat(P1,2,2);

VarConst(2,:) = convert4store(A'*P);
expr = [expr,'v2=At*v1;'];

function expr = atv21(expr)
%Var1 = A'Var2

global VarConst

A1 = VarConst(3,:);
A = store2mat(A1,2,2);

P1 = VarConst(2,:);
P = store2mat(P1,2,2);

VarConst(1,:) = convert4store(A'*P);
expr = [expr,'v1=At*v2;'];

function expr = atv22(expr)
%Var2 = A'Var2
```

```
global VarConst

A1 = VarConst(3,:);
A = store2mat(A1,2,2);

P1 = VarConst(2,:);
P = store2mat(P1,2,2);

VarConst(2,:) = convert4store(A'*P);
expr = [expr,'v2=At*v2;'];

function expr = pav11(expr)
%Var1 = Var1*A

global VarConst

A1 = VarConst(3,:);
A = store2mat(A1,2,2);

P1 = VarConst(1,:);
P = store2mat(P1,2,2);

VarConst(1,:) = convert4store(P*A);
expr = [expr,'v1=v1*A;'];

function expr = pav12(expr)
%Var1 = Var2*A

global VarConst

A1 = VarConst(3,:);
A = store2mat(A1,2,2);

P1 = VarConst(2,:);
P = store2mat(P1,2,2);

VarConst(1,:) = convert4store(P*A);
expr = [expr,'v1=v2*A;'];

function expr = pav21(expr)
%Var2 = Var1*A

global VarConst

A1 = VarConst(3,:);
A = store2mat(A1,2,2);

P1 = VarConst(1,:);
P = store2mat(P1,2,2);

VarConst(2,:) = convert4store(P*A);
expr = [expr,'v2=v1*A;'];

function expr = pav22(expr)
```

```
%Var2 = Var2*A

global VarConst

A1 = VarConst(3,:);
A = store2mat(A1,2,2);

P1 = VarConst(2,:);
P = store2mat(P1,2,2);

VarConst(2,:) = convert4store(P*A);
expr = [expr,'v2=v2*A;'];

function expr = subt1(expr)
%Var1 = Var1 - Var2

global VarConst

P1 = VarConst(1,:);
P2 = VarConst(2,:);

VarConst(1,:) = P1 - P2;
expr = [expr,'v1=v1-v2;'];

function expr = subt2(expr)
%Var2 = Var2 - Var1

global VarConst

P1 = VarConst(1,:);
P2 = VarConst(2,:);

VarConst(1,:) = P1 - P2;
expr = [expr,'v2=v2-v1;'];

function expr = subt3(expr)
%Var1 = Var1 - Var2

global VarConst

P1 = VarConst(1,:);
P2 = VarConst(2,:);

VarConst(2,:) = P1 - P2;
expr = [expr,'v2=v1-v2;'];

function expr = subt4(expr)
%Var2 = Var2 - Var1

global VarConst

P1 = VarConst(1,:);
P2 = VarConst(2,:);
```

```
VarConst(1,:) = P1 - P2;
expr = [expr,'v1=v2-v1;'];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Validation Test for the evolved program
%Compare with results generated by normal methods
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function results = testresults(individual,Q)
global VarConst

Functions = ['addi1';'addi2';'addi3';'addi4';'subt1';'subt2';'subt3';'subt4'; ...
      'atv11';'atv12';'atv21';'atv22';'pav11';'pav12';'pav21';'pav22'];


numvariables = 2;        %define number of variables the run going to use
numconstants = 2;                      %define number of constants here

T = 0.9*rand(2,2)
A = T*[0.5 -0.5; 1 0]*T'

Variables = ones(2,4);
Constants = ones(2,4);
VarConst = [Variables;Constants];
VarConst(3,:) = convert4store(A);
VarConst(4,:) = convert4store(Q);


results =[];

[rr,endnode] = find(individual==9999);
expression = [''];

[results,espression]=evalindividual(individual);
disp('results of P from GP');
results

Pk = [1 1; 1 1];

for i = 1:30
   Pk_1 = (A')*Pk*A + Q;
   if abs(Pk_1-Pk)<0.05
      i
      break;

   end

   Pk = Pk_1;
end
disp('results from iteration')
Pk
```

# Appendix B

# Matlab Code for Self-Evolved MRAS

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Function GPmain is the main program that calls all other programs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [Individual, bestind]=gpmain()

global Functions Geneoperations MaxDepth %genepoolfitness genepoolresults
global Stopflag matrow matcol population ADFrecord ADFDepth
global VarConst numvariables numconstants Stopflag
global Individual IndividualDepth IndividualFitness
global uc ym y u tuc tuc1 tuc2 sy sy1 sy2 ru1

population = 50;                %define population size
maxstring = 150;                %define the maximum length of the string to
%describe the operations of the genome
maxdepth = 50;               %define the maximum depth of the genome
numgenerations = 2;          %define the number of generations for the run
matrixspace = 500;           %define the amount of matrix space to allocate
%will be used as memory for variables
Stopflag = 0;                %Stopflag, global flag for termination criteria
bestfit = 0;                 %define initial best fitness

IndividualDepth = 3500;   %definfe the depth and complexity of each individual
noplayers =10;            %define number of players in tournament game


results = []; varindex = []; espression = []; tmpfit = zeros(population,1);

format long;

%Define functions and genetic operations
%enter function names here, but name must be 5 letters

Functions = ['uca00';'uc1s0';'ys000';'y1a00';'u1a00';'ucs00'; ...
```

```
      'uc1a0';'ya000';'y1s00';'u1s00';'uca0b';'uc1sb';'ys00b';'y1a0b';...
      'u1s0b';'u1a0b';'ucs0b';'uc1ab';'ya00b';'y1s0b';'nothi'];
%enter gene operations here, but name must be 5 letters
Geneoperations = ['cross'; 'mutat'; 'repro'];   %name must be 5 letters

%initialise initial population
initvariables;
initIndividual(population);
%Individual
%evaluate initial fitness of individuals
for pop = 1:population
   if Stopflag == 0
      initvariables;
      tmpfit(pop) = evalfitind(Individual(pop,:));
   end
end

%Individual
tmpfit
tmpmax = 0;
for gen = 1:80
   if Stopflag == 0
      tmpNewGen=zeros(population,IndividualDepth);
      tmpfit = zeros(population,1);

      NewPop = 1;
      while NewPop <= population

         chooseop = ceil(10*rand(1));
         if ((chooseop>1)&(chooseop<=3))
            NewInd = repro(noplayers);
            tmpNewGen(NewPop,:)=NewInd;
            tmpfit(NewPop) = evalfitind(NewInd);

            %NewPop = NewPop+1;
         elseif chooseop == 1
            NewInd = mutat(noplayers);                  %new mutant
            tmpNewGen(NewPop,:)=NewInd;
            tmpfit(NewPop) = evalfitind(NewInd);

         else
            [NewInd1, NewInd2] = cross(noplayers);
            tmpNewGen(NewPop,:)=NewInd1;
            tmpfit(NewPop) = evalfitind(NewInd1);

            NewPop = NewPop+1;
            if NewPop <= population
               tmpNewGen(NewPop,:)=NewInd2;
               tmpfit(NewPop) = evalfitind(NewInd2);
            else
            end

         end
      end
      IndividualFitness = tmpfit;
      Individual = tmpNewGen;
      [mm,imax]=max(IndividualFitness);

      if mm > tmpmax
         bestind = Individual(imax,1:IndividualDepth);
         initvariables;
         fitness = evalfitindresults(bestind);
         tmpmax = mm
         genrec = gen
      else
      end
      disp('one generation');

   end
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Genetic Operations: Reproduction and Crossover
%Mutation operation included.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function NewInd = repro(noplayers)
%reproduces the same genome for the next generation
global Individual

index = selectindividual(noplayers);

NewInd = Individual(index,:);

function [NewInd1, NewInd2] = cross(noplayers)
%choose a parent according to the probability of the fitness of the genepool
%do cross over and return the New Individual

global Individual

sizeind = size(Individual,2);
NewInd1 = zeros(1,sizeind);
NewInd2 = zeros(1,sizeind);
index1 = 1; index2 = 1;

while index1==index2
   index1 = selectindividual(noplayers);
   index2 = selectindividual(noplayers);
end

parent1 = Individual(index1,:);
parent2 = Individual(index2,:);

[tmpfragment1, A1,A2] = getfrag(parent1);
[tmpfragment2, B1,B2] = getfrag(parent2);
```

```
[Row,P1end] = find(parent1 == 9999);            %get the last node
NewInd1frag = [parent1(1,(1:A1-1)), tmpfragment2, parent1(1, A2+1:P1end)];
[Row,P2end] = find(parent2 == 9999);            %get the last node
NewInd2frag = [parent2(1,(1:B1-1)), tmpfragment1, parent2(1, B2+1:P2end)];

NewInd1(1:size(NewInd1frag,2))= NewInd1frag;
NewInd2(1:size(NewInd2frag,2))= NewInd2frag;

function NewInd1 = mutat(noplayers)

global Individual IndividualDepth Functions

index = ceil(rand(1,1)*size(Individual,1));
sizeofFunction = size(Functions,1);
parent1 = Individual(index,:);

sizeind = size(Individual,2);
NewInd1 = zeros(1,sizeind);


[tmpfragment1, A1,A2] = getfrag(parent1);
tmpfragment2 = zeros(1,(A2-A1+1));
for i = 1:(A2-A1+1)
   tmpfragment2(1,i) = ceil(rand(1,1)*(sizeofFunction));
end


[Row,P1end] = find(parent1 == 9999);            %get the last node
NewInd1frag = [parent1(1,(1:A1-1)), tmpfragment2, parent1(1, A2+1:P1end)];
NewInd1(1:size(NewInd1frag,2))= NewInd1frag;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Evaluation of Individuals
%Calcalate results and fitness of individuals.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function fitness = evalfitind(individual)
%return the fitness of one individual

global Stopflag matrow matcol uc ym y u uc

%tmpu = zeros(2000,1);
error = zeros(2000,1);
inverr = zeros(2000,1);
errdt = zeros(2000,1);

espressions = [];

initvariables;
coef =evalindividual(individual);


for t = 3:300
```

```
   ym(t,1) = 0.0182*uc(t-1,1)+0.01657*uc(t-2,1)+1.721*ym(t-1,1)-0.7558*ym(t-2,1);
   y(t,1) =1.96*y(t-1,1)-y(t-2,1)+0.002791*u(t-1,1)+0.002791*u(t-2,1);
   u(t,1) = coef(1,1)*uc(t,1)+coef(2,1)*uc(t-1,1)+coef(4,1)*y(t,1)+ ...
      coef(5,1)*y(t-1,1)+coef(7,1)*u(t-1,1);

   error(t,1) = abs(y(t,1) - ym(t,1));
   errdt(t,1) = abs(error(t,1)-error(t-1,1));

end
%figure;hold on;
%plot(u);
%plot(ym);
%hold off;

fitness = 1/(1+(4*sum(errdt))+sum(error));  %4 is an arbitrary weight
%but not really necessary.

%fitness = 1/(1+sum(errdt));
%fitness = 1/(1+sum(error));


function [Tmpgenestring, nA, nB]= getfrag(genomeA)
%for crossover, the function return a fragment
% of the genome to be used for the crossover
%operation.

nodeA1 = 1;
nodeA2 = 1;

nodeA1 = pickanode(genomeA);
nodeA2 = pickanode(genomeA);

if nodeA1 == nodeA2
   nA = nodeA1;
   nB = nodeA2;
   Tmpgenestring = genomeA(nodeA1);
else

   nodeA = [nodeA1, nodeA2];
   nodeA = sort(nodeA);         %do a sort so that node A1 < node A2
   nA = nodeA(1,1);
   nB = nodeA(1,2);
   Tmpgenestring = genomeA(:,nA:nB); %the fragment is [func,B,C,...,C]
end


function genomerow = convert4store(genemat)
%convert a (r x c) matrix to a row vector (of size [1,r x c])to
% be used for
%storage of genepool. see also 'store2mat'.
genomerow = [];
[row,col]=size(genemat) ;
for r = 1:row
   for c = 1:col
```

```
      genomerow = [genomerow genemat(r,c)];
   end
end

function genemat = store2mat(genomerow,row,col)
%convert a row vector (of size [1,r x c]) genomerow to a (r x c) matrix
%genemat to be used for evaluation. see also 'convert4store'.

s = 1;
genecolsize = size(genomerow,2);
givensize = row*col;
if givensize == genecolsize
   for r = 1:row
      for c = 1:col
         genemat(r,c) = genomerow(s);
         s = s + 1;
      end
   end
else
   disp('row and col size does not match dimension of input');
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Selection of individaul programs for genetic operations
%via Tournament method
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function index = selectindividual(noplayers)
%select individuals from genepool via tournament methods
%returns the index of the individual from Individualfitness

global IndividualFitness population

indfit = zeros(noplayers,1); bestfit = 0;

tournamentpool = ceil(population*rand(noplayers,1));    %index of the players
%noplayers
%tournamentpool
for player = 1:noplayers
   indfit(player)= IndividualFitness(tournamentpool(player,1));
   if bestfit <= indfit(player)
      bestfit = indfit(player);
      index = tournamentpool(player,1);
   else
      index = tournamentpool(1,1);

   end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Initialisation of Individuals and Variables
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function initvariables()
%initialise number of variables and constants here
global uc ym y u tuc tuc1 tuc2 sy sy1 sy2 ru1

tuc = 0; tuc1 = 0; tuc2 = 0; sy = 0; sy1 = 0; sy2 = 0; ru1 = 0;

uc = zeros(2000,1);ym = zeros(2000,1);y = zeros(2000,1);u = zeros(2000,1);
uc(50:200,1) = 1;

function initIndividual(pop)
%function initialise a number of individuals
global Individual IndividualDepth IndividualFitness Functions

sizeofFunction = size(Functions,1);
%sizeofADFrec = size(ADFrecord,1);
Individual = zeros(pop,IndividualDepth);

for i=1:100

Individual(1:pop,i) = ceil(rand(pop,1)*(sizeofFunction));

end

Individual(1:pop,i+1) = 9999;

IndividualFitness = zeros(pop,1);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Functions which can be used by the Genetic Programming System
%to Evolve its solution.
%Boring but necessary
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function cof = u1a00(cof)

cof(7,1) = cof(7,1) + 1;

function cof = u1a0b(cof)

cof(7,1) = cof(7,1) + 0.1;

function cof = u1s00(cof)

cof(7,1) = cof(7,1) - 1;

function cof = u1s0b(cof)

cof(7,1) = cof(7,1) - 0.1;

function cof = uc1a0(cof)

cof(2,1) = cof(2,1) + 0.1;
```

```
function cof = uc1ab(cof)

cof(2,1) = cof(2,1) + 1;

function cof = uc1s0(cof)

cof(2,1) = cof(2,1) - 0.1;

function cof = uc1sb(cof)

cof(2,1) = cof(2,1) - 1;

function cof = uca00(cof)

cof(1,1) = cof(1,1) + 0.1;

function cof = uca0b(cof)

cof(1,1) = cof(1,1) + 1;

function cof = y1a00(cof)

cof(5,1) = cof(5,1) + 0.1;

function cof = y1a0b(cof)

cof(5,1) = cof(5,1) + 1;

function cof = y1s00(cof)

cof(5,1) = cof(5,1) - 0.1;

function cof = y1s0b(cof)

cof(5,1) = cof(5,1) - 1;

function cof = ya000(cof)

cof(4,1) = cof(4,1) + 0.1;

function cof = ya00b(cof)

cof(4,1) = cof(4,1) + 1;

function cof = ys000(cof)

cof(4,1) = cof(4,1) - 0.1;

function cof = ys00b(cof)

cof(4,1) = cof(4,1) - 1;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Display of Evolved Solution
%Control signal, Ym and Y output of process
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function fitness = evalfitindresults(individual)
%return the fitness of one individual

global Stopflag matrow matcol uc ym y u uc

%tmpu = zeros(2000,1);
error = zeros(2000,1);
errdt = zeros(2000,1);

espressions = [];

initvariables;
coef =evalindividual(individual)


for t = 3:600
   ym(t,1) = 0.0182*uc(t-1,1)+0.01657*uc(t-2,1)+1.721*ym(t-1,1)-0.7558*ym(t-2,1);
   y(t,1) =1.96*y(t-1,1)-y(t-2,1)+0.002791*u(t-1,1)+0.002791*u(t-2,1);
   u(t,1) = coef(1,1)*uc(t,1)+coef(2,1)*uc(t-1,1)+coef(4,1)*y(t,1)+ ...
      coef(5,1)*y(t-1,1)+coef(7,1)*u(t-1,1);
   error(t,1) = abs(y(t,1) - ym(t,1));
   errdt(t,1) = abs(error(t,1)-error(t-1,1));
end
figure;
plot(u);
figure;hold on;
plot(ym);
plot(y,'r');
hold off;

totalerror = sum(error)

fitness = 1/(1+sum(error))
```