University of Southern Queensland

Faculty of Engineering & Surveying

PID Controller Optimisation Using Genetic Algorithms

A dissertation submitted by

Matthew Robert Mackenzie

in fulfilment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Engineering (Computer Systems)

Submitted: October, 2007

Abstract

Genetic Algorithms are a series of steps for solving an optimisation problem using genetics as the model (Chambers, 1995). More specifically, Genetic Algorithms use the concept of Natural Selection – or *survival of the fittest* – to help guide the selection of candidate solutions. This project is a software design-and-code project with the aim being to use MATLAB® to develop a software application to optimise a Proportional-Integral-Derivative (PID) Controller using a purpose built Genetic Algorithm as the basis of the optimisation routine. The project then aims to extend the program and interface the Genetic Algorithm optimisation routine with an existing rotary-wing control model using MATLAB®.

A systems approach to software development will be used as the overall framework to guide the software development process consisting of the five main phases of Analysis, Design, Development, Test and Evaluation.

The project was only partially successful. The Genetic Algorithm did produce reasonably optimal values for the PID parameters; however, the processing time required was prohibitively long. Additionally, the project was unsuccessful in interfacing the optimised controller to the existing rotary-wing model due difficulty in conversion between SIMULINK® and MATLAB® formats. Further work to apply code optimisation techniques could see significant reduction in processing times allowing more iterations of the program to execute thereby achieving more accurate results.

Thus the project results suggest that the use of Genetic Algorithms as an optimisation method is best suited to complex systems where classical optimisation methods are impractical.

Certification

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Matthew Robert Mackenzie

Student Number: 001323707

Signature

Date

Acknowledgements

I would like to thank my supervisor Paul for guiding me through this challenge.

I would also like to thank my wife Sandra whose patience has allowed us to balance this project, work and our young family.

My love to my wife Sandra, my eldest son Joseph and my youngest son Alec.

Matthew Mackenzie

University of Southern Queensland
October 2005

Contents

List of Figures

List of Tables

Glossary of Genetic Terms

Allele	1. The American Heritage® Dictionary (2004) defines an allele as "one member of a pair or series of genes that occupy a specific position on a specific chromosome."
Chromosome	1. The American Heritage® Stedman's Medical Dictionary (2006) defines a chromosome as "a threadlike linear strand of DNA and associated proteins in the nucleus of eukaryotic cells that carries the genes and functions in the transmission of hereditary information."
Gene	2. The American Heritage® Stedman's Medical Dictionary (2006) defines gene as "a hereditary unit that occupies a specific location on a chromosome, determines a particular characteristic in an organism by directing the formation of a specific protein, and is capable of replicating itself at each cell division." 3. The American Heritage® Dictionary (2004) defines a gene as "a hereditary unit consisting of a sequence of DNA that occupies a specific location on a chromosome and determines a particular characteristic in an organism. Genes undergo mutation when their DNA sequence changes."

Genetic Algorithm	4. Whitley (n.d.) defines Genetic Algorithms as “a family of computational models inspired by evolution” which are also “population-based models that uses selection and recombination operators to generate new sample points in a search space.” 5. Cantu-Paz (2001) defines Genetic Algorithms as “stochastic search algorithms based on principles of natural selection and genetics. Genetic Algorithms attempt to find good solutions to the problem at hand by manipulating a population of candidate solutions.” 6. Haupt and Haupt (2004) defines Genetic Algorithms as “an optimization and search technique based on the principles of genetics and natural selection.” 7. Chambers (1995) defines Genetic Algorithms as “a problem-solving method that uses genetics as its model of problem solving.”
Kinetochore	8. Merriam-Webster’s Medical Desk Dictionary (2002) explains that the Kinetochore is “… the point or region on a chromosome to which the spindle attaches during mitosis” and that the Kinetochore is also called the centromere. 9. Reproductive cells divide at a random point along the chromosome known as the kinetochore (Haupt & Haupt 2004, sec. 1.4).

Meiosis	10. The American Heritage® Dictionary (2004) defines meiosis as "…the process of cell division in sexually reproducing organisms that reduces the number of chromosomes in reproductive cells from diploid to haploid, leading to the production of gametes in animals and spores in plants." 11. Crossing mimics the genetic process of meiosis that results in cell division (Haupt & Haupt 2004, sec. 1.4). 12. The process of cell division for higher [multiple cell] organisms is called meiosis (Haupt & Haupt 2004, sec. 1.3). Whereas, cell division for simple single-celled organisms is called mitosis (Haupt & Haupt 2004, sec. 1.3).
Mutation	13. The American Heritage® Stedman's Medical Dictionary (2006) defines mutation as "the process by which such a sudden structural change occurs, either through an alteration in the nucleotide sequence of the DNA coding for a gene or through a change in the physical arrangement of a chromosome". 14. The American Heritage® Dictionary (2004) defines mutation as "a change of the DNA sequence within a gene or chromosome of an organism resulting in the creation of a new character or trait not found in the parental type".
Natural Selection	15. Natural Selection is the process that occurs in nature whereby the strongest organisms, in terms of fitness for their environment live to reproduce more often and successfully, thus passing on their genetic traits to their offspring and making their genetic traits more prolific in the species.

Selection	16. Merriam-Webster's Medical Desk Dictionary (2002) defines selection as "a natural or artificial process that results or tends to result in the survival and propagation of some individuals or organisms but not of others with the result that the inherited traits of the survivors are perpetuated." 17. The American Heritage® Dictionary (2004) defines selection as "a natural or artificial process that favors or induces survival and perpetuation of one kind of organism over others that die or fail to produce offspring".

Chapter 1

Introduction

1.1 Project Description

Genetic Algorithms are a series of steps for solving an optimisation problem using genetics as the underpinning model (Chambers, 1995). More specifically, Genetic Algorithms use the concept of Natural Selection – or *survival of the fittest* – to help guide the selection of candidate solutions. In essence, Genetic Algorithms use an iterative process of selection, recombination, mutation and evaluation in order to find the fittest candidate solution [Haupt and Haupt (2004), Whitley (n.d.) and Chambers (1995)]. This project is a software design-and-code project with the aim being to use MATLAB® to develop a software application to optimise a PID Controller using a purpose built Genetic Algorithm as the basis of the optimisation routine.

The core of the project is the research, design, coding and testing of the Genetic Algorithm optimisation program. However, the project will then attempt to interface the Genetic Algorithm optimisation routine with an existing rotary-wing control model using MATLAB®. This interface will first require the conversion of the existing model in SIMULINK® to a MATLAB® construct.

Without the use of a Genetic Algorithm, the PID Controller would rely upon classical analytical optimisation techniques. Such techniques are best suited to problems with only a few variables because of the need to develop a mathematical model of the system from which the use of derivatives can be used to find the optimal solution. In comparison, a Genetic Algorithm can handle multiple variables and only requires the ability to develop a mathematical model to configure a set of inputs (the *variables*) in order for the model to produce an optimal output (the *cost*).

Hence a PID Controller with three main variables – normally denoted as q_0, q_1 and q_2 – is ideally suited to using a Genetic Algorithm to optimise the controller's response as it is a multi-variable system and it has well understood and proven cost functions, such as Integral Time $\times$ Absolute Error (ITAE), Integral Absolute Error (IAE) and Integral Squared Error (ISE).

1.2 Aims and Objectives

The aim of this project is to use MATLAB® to design-and -code an optimised PID controller using a Genetic Algorithm to perform the optimisation routine.

The aim is then broken down to establish four primary and two secondary objectives for the project:

Primary Objectives (Base Functionality)

Step 1. Research the background information relating to Genetic Algorithms.

Step 2. Design a Genetic Algorithm for implementation using a 3rd generation program language, specifically MATLAB®, within set specifications (refer Appendix A Project Specification).

Step 3. Code the designed Genetic Algorithm using MATLAB®.

Step 4. Test the Genetic Algorithm against specifications.

Secondary Objectives (Advanced Functionality)

Step 5. Increase the functionality of the Genetic Algorithm through the addition of a user option to configure for Roulette Wheel based selection.

Step 6. Model the Genetic Algorithm for use controlling a rotary-wing control system using MATLAB® (SIMULINK® rotary-wing model to be provided by the Project Supervisor).

1.3 Dissertation Overview

This dissertation is structured into two main parts. The first part provides the background to the project by explaining the goals and objectives along with a review of background literature pertaining to Genetic Algorithms. The dissertation then discusses optimisation and proposes a categorisation system in order to assist with determining what optimisation problems are best suited to be solved by Genetic Algorithms. A brief review of digital controllers is provided in order to set the context for the project itself and explain the importance of finding the optimum values for the three PID parameters – q_0, q_1 and q_2. This first part concludes with a detailed discussion of Genetic Algorithms including their theory and operation.

Chapter One – Introduction, including objectives, background literature review and project methodology

Chapter Two – Optimisation

Chapter Three – Digital Controllers

Chapter Four – Genetic Algorithms

The second part of the dissertation describes the Genetic Algorithm designed and coded in MATLAB® to solve the project problem – named ***Argo***[1]. After describing how the application is structured, the results of the optimised PID controller are then presented. Finally, the dissertation summarises the project's goals, objectives and results before suggesting how the project could be extended for future projects.

Chapter Five – ***Argo***

Chapter Six – Analysis & Results

Chapter Seven – Conclusion

[1] The Argo was the ship, built by Argos with the help of Athena, in which Jason and the Argonauts sailed in quest of the Golden Fleece. It was the largest ship ever built, and its crew included Heracles, Orpheus, and a host of other heroes from all over Greece. Athena fitted the bow of the ship with a speaking timber, cut from the sacred oaks of Dodona (Hunter, 2002).

1.4 Background Literature Review

The first main task of this project involved the research of background literature on Genetic Algorithms. Genetic Algorithms are computer based processes for which optimisation of a problem is achieved by mimicking nature's own process of Natural Selection – also referred to as *survival of the fittest* (Buckland, 2005a). Although the subtleties of the definition of Genetic Algorithms vary, the intent is the same across all researched sources.

Although all sources provide ostensibly the same basic definition for a Genetic Algorithm, there are variances associated with terminology. However, these differences are often made with reference to slightly different concepts. For example, Haupt and Haupt (2004, sec. 2.1) refer to Binary Genetic Algorithms. This terminology highlights that the particular Genetic Algorithms described are first coded in binary before further operation. Although some Genetic Algorithms require encoding for use on a computer, the method of encoding can vary. Whitley (n.d.) provides another example of variances in terminology by consistently referring to the Canonical Genetic Algorithm, in order to baseline the discussion by establishing a standard or basic form of Genetic Algorithm from which to later extend upon.

Whitley (n.d.) notes that research on Genetic Algorithms is generally first credited to John Holland (1975), with substantial work following thereafter by students of his such as DeJong (1975). Thus, it can be surmised that with such a short life thus far, the field of Genetic Algorithms is still maturing. Indeed, whilst there exists substantial background literature on the field of Genetic Algorithms itself, very few sources make significant contributions to the area of Genetic Algorithm Applications. An important exception is the work presented by Chambers (ed. 1995) with much research and compilation of practical Genetic Algorithms.

In a similar manner, although many sources [such as Whitley (n.d.), AAAI (2000-2005) and Cantu-Paz (2001)] describe and explain Genetic Algorithms, working examples coded for practical use are minimal. Of note, Haupt and Haupt (2004), Chambers (1995) and Buckland (2005a) are important exceptions providing many valuable generic examples that can be used by the reader to code a Genetic Algorithm for a practical application. Indeed, Buckland (2005a) provides one of the only complete, but simple, examples of a coded Genetic Algorithm within the researched literature.

1.4.1 Existing Research Emphasis

Interestingly, although most sources provide a description of what Genetic Algorithms are and how they are structured, very few provide the important rationale and justification for why a Genetic Algorithm would be used and when it would be best applied. However, Haupt and Haupt (2004) provide a full and detailed introduction to Genetic Algorithms. Importantly, Haupt and Haupt (2004) make the key insight that Genetic Algorithms are used to solve optimisation problems. Whitley (n.d.) extends on this insight noting that Genetic Algorithms are useful for solving parameterised optimisation problems. Further, Haupt and Haupt (2004) then extend on this to categorise all optimisation problems and identify those categories that are most suited to Genetic Algorithms and which categories are more suited to classical optimisation techniques.

Generically, the Genetic Algorithm is basically on an iterative process of selection, recombination, mutation and evaluation [Haupt and Haupt (2004), Whitley (n.d.) and Chambers (1995)]. In defining the Canonical Genetic Algorithm, Whitley (n.d.) distinguishes between the evaluation and fitness functions; within the sub-process of evaluation. In this respect the evaluation function is independent of evaluation of other chromosomes, whereas fitness is defined with respect to other members of the population. Haupt and Haupt (2004) also distinguish between the terms fitness and cost, whereby the goal of a Genetic Algorithm is to locate a chromosome with maximum fitness, or, minimum cost – used dependent upon the nature of the problem at hand.

One of the key benefits that Genetic Algorithms have over conventional analytical based methods is the ability to find the global maxima/minima. This is achieved even if the problem space contains numerous local maxima/minima. However, dependent upon how the Genetic Algorithm is constructed, often the selection and crossover algorithms may be too effective. The result of this can be a population with broadly similar characteristics – the very feature that Genetic Algorithms need not to posses in order not to converge on local maxima/minima. Wall (n.d.) proposes that this problem – referred to as DeJong-style crowding – can be mitigated against by using a replace-most-similar replacement scheme. Wall (n.d.) also proposes another method for maintaining diversity within the population by using a Goldberg style fitness scaling method.

Chambers (1995) and Haupt & Haupt (2004) present a number of ways to encode the parameters in order to use a Genetic Algorithm to optimise the problem, such as binary or Gray coding. However, it is Chambers (1995) that argues that a continuous Genetic Algorithm is superior. Chambers (1995) argues that there is no need to code a Genetic Algorithm's parameter as the Genetic Algorithm can be designed to work with continuous variables. By working with continuous variables a performance gain is achieved immediately as there is no requirement to calculate a conversion from continuous to binary. But perhaps the major advantage of using a continuous Genetic Algorithm is the avoidance of the problem of selecting the number of bits from which to represent the variable (Chambers, 1995). Chambers (1995) also cites Michalewicz (1992) to further support his argument noting that his "conducted experiments indicate that the floating point representation is faster, more consistent from run to run, and provides a higher precision (especially with large domains where binary coding would require prohibitively long representations)."

Genetic Algorithms are a natural optimisation technique; based on the process of Natural Selection (Whitley, n.d.). Haupt and Haupt (2004) provide a number of other natural optimisation techniques including simulated annealing, particle swarm optimization, ant colony optimization and evolutionary algorithms. Chambers (1995) goes further to provide an overview of each technique as well as cultural algorithms.

Genetic Algorithms provide a means for providing solutions to complex optimisation problems [Whitley (n.d.), Haupt and Haupt (2004), Chambers (1995) and Whitley (n.d.)]. However, Cantu-Paz (2001) suggests that Genetic Algorithms are likely to only provide a reasonably good solution given a reasonable limit to the processing time available. Indeed, dependent upon the accuracy required and the processing cost of the evaluation function, Genetic Algorithms may even take years to find an acceptable solution (Cantu-Paz, 2001). Reducing the processing time of Genetic Algorithms is the motive behind Cantu-Paz (2001) work on producing a parallel implementation of a Genetic Algorithm.

Reinforcing Cantu-Paz's work, Garrido et al. (n.d) propose the adaptation of a Genetic Algorithm to result in predictive control using a technique referred to as Restricted Genetic Optimisation (RGO). Unlike conventional Genetic Algorithms, RGO does not search the entire solution space to generate the next generation, but rather searches only in a point neighbourhood around the best solution. RGO does perform a global search at the beginning and then local searches thereafter. Carrido et. al. (n.d.) explains that new solutions are oriented in the direction of the steepest slope of the cost function with solutions restricted to points within a radius proportional to uncertainty.

Importantly, Garrido et al. (n.d., ch. 1, pp. 1-3) make the key insight that stochastic optimisation methods (such as Genetic Algorithms) may be well suited to time varying functions with noise, such a control systems. This is because noisy systems are non-differentiable when modelled mathematically. Practically, noisy systems are often optimised by ignoring the impact of noise. This technique is problematic when dynamic optimisation is desired.

1.4.2 Future Research Areas

This review has discussed the background to the major elements of Genetic Algorithms. However, it is important to note the areas that require further research within the field. In doing so, this review makes the observation that there is an apparent need for further experimental efforts to measure the actual performance improvement of Genetic Algorithms over classical analytical techniques. Although there are many statements made by authors indicating the benefits of using a Genetic Algorithm for optimisation problems, there needs to be research performed to measure the improvements across the various different categories – perhaps using Haupt and Haupt's (2004) categorisation scheme as a basis. Also, perhaps that research would discover the crossover point at which a Genetic Algorithm becomes more efficient and/or successful than a classical technique. For example, experimentation could propose the number of variables at which point the problem is more efficient to be solved using a Genetic Algorithm. Whilst it is unlikely that a single set of parameter values would be uncovered suitable for all problems, there would be value in identifying a set of *rules of thumb* that could be applied to optimisation problems.

1.4.3 Literature Review Summary

In summary, this literature review has attempted to canvas the background literature available on Genetic Algorithms. In doing so, this review has noted that the field of Genetic Algorithms is still maturing, and therefore there is still much research yet to be conducted within the field. This review has suggested that the area within the field that most requires further research is in measuring a Genetic Algorithm's improvements over classical optimisation techniques and identifying what types of problems are more suited to Genetic Algorithms.

It is noted however, that there is much detailed research on the basic topic of Genetic Algorithms including how they function and their basis on nature. Within this body of research there are many sources that provide examples of how elements of a Genetic Algorithm would be coded.

1.5 Project Methodology

In order to appreciate the project results, it is important to understand how the project was practically undertaken. Thus the project methodology is explained in the following sections.

1.5.1 Systems Approach to Software Development

A systems approach to software development was used as the overall framework to guide the software development process (refer Figure 1-1). The systems approach to software development consist of five main phases:

Step 1. **Analysis**. Analyse the problem and define the requirements.

Step 2. **Design**. Design the structure of program including functions and interfaces.

Step 3. **Development**. Code all functions.

Step 4. **Test**. Test all functions and program perform to specification.

Step 5. **Evaluation**. Evaluate the performance of the program and confirm it achieves overall aim.

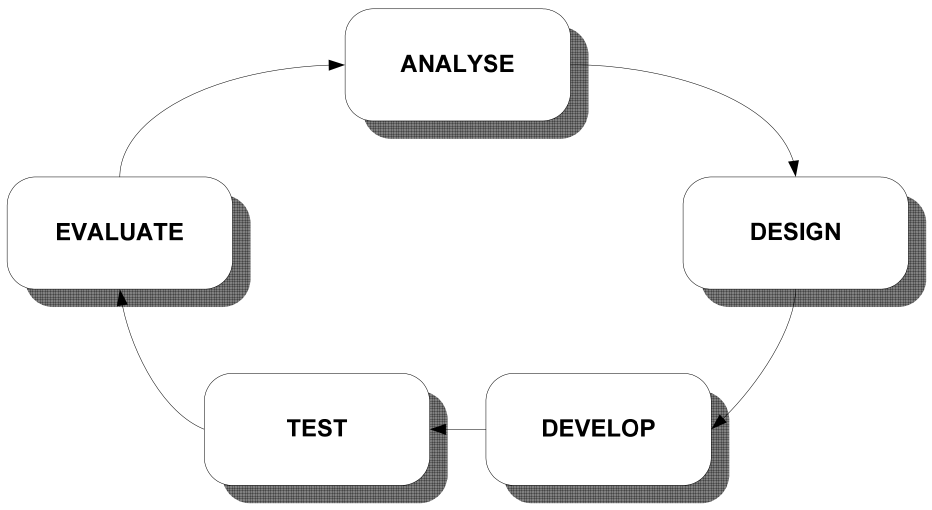


Figure 1-1 Systems Approach

In practice, each of the phases can overlap and often require iterations at each stage. The overall process can also be conducted iteratively as the results of evaluation are rolled back into the development process.

The systems approach was used vice a traditional waterfall approach because of the need for iteration within the model for improvement and extension (discussed further at 1.5.4).

1.5.2 Programming Models

There are a number of different programming models available today such as CASE tools, Integrated Development Environments and object oriented programming. However, MATLAB® was chosen as the programming language for two main reasons: firstly, familiarity with the tool and procedural programming by the developer; and secondly, suitability of the program for a medium size and medium complexity programming task.

1.5.3 Test Program

Once the software was coded the next step was the testing program. This project adopted a simple two-phase test program.

Step 1. **Unit Level.** The first phase is unit level testing. Each function is tested in isolation in order to control its environment; specifically the inputs and interfaces. Each unit will be tested for normal function operation,

operation at the limits of the function inputs, and non-normal function inputs.

Step 2. **System Level.** The second phase is system level testing. The system – ***Argo*** – will be tested as a whole; that is, all functions correctly interfaced. The system will be tested for normal system operation, operation at the limits of the system inputs, and non-normal system inputs.

Practically, in order to test the program's operation, a test control system will be used to allow the Genetic Algorithm to optimise the values of the PID parameters. Two-test control systems will be used – one based on a first order system and the other based on a second order system. The Project Supervisor provided the first order test control system. The second order test control system was a digital control system used as part of an assignment for the University of Southern Queensland course ELE3105 Computer Control Systems (Mackenzie, 2004).

The optimal values of the PID parameters are known for both the test control systems (calculated using the in-built MATLAB® function *fminsearch*). The known optimal values acted as the baseline from which the Genetic Algorithm's results will be compared against to confirm successful operation.

1.5.4 Evaluation and Extension

The evaluation phase of the systems approach to software development was conducted to confirm that normal operation had been successfully achieved and all specifications were met.

Once the core of the program was operational, a second iteration of the systems approach will be conducted again to meet the secondary objectives as set out in Appendix A Project Specification.

1.6 Summary

In summary, this project is a software design and code project with the aim being to use MATLAB® to develop a software application to optimise a PID Controller using a purpose built Genetic Algorithm as the basis of the optimisation routine. The core of the project is the research, design, coding and testing of the Genetic Algorithm optimisation program. However, the project will then attempt to interface the Genetic Algorithm optimisation routine with an existing rotary-wing control model using MATLAB®.

In developing the necessary software, the fundamental development philosophy used was a simple systems approach to software development. This approach was performed initially to confirm that the basic Project Specification requirements have been met, and was then performed again to meet the secondary objectives.

Practically, in order to test the programs operation, a test control system will be used to allow the Genetic Algorithm to optimise the values of the PID parameters of the test control system. The optimal values of the PID parameters are known for the test control system and will be used as the baseline from which the Genetic Algorithm's results will be compared against to confirm successful operation.

Chapter 2

Optimisation

2.1 Introduction

Genetic Algorithms are computer based processes for which optimisation of a problem is achieved by mimicking nature's own process of Natural Selection – also referred to as *survival of the fittest* (Buckland, 2005a). Before the concept of Genetic Algorithms can be studied in detail, it is relevant to review the concept of optimisation itself, and propose a simple model for optimisation that can be used to better understand what is required of any optimisation routine.

In doing so, this chapter will first present a generic model for optimisation problems and then compare the mathematical process of root finding with optimisation for completeness. The chapter will then present a categorisation scheme for optimisation problems in order to help identify which problems may be suited to using a Genetic Algorithm. The chapter will conclude with a brief review of other natural optimisation methods.

2.2 Optimisation Models

Expanding upon Whitley (n.d., p. 2), a generic model for optimisation can be viewed as the configuration of a set of parameters, variables or characteristics (*the inputs*) in order for the function, model or experiment (*the process*) to produce an optimal cost, objective or result (*the output*). Figure 2-1 graphically represents this model.

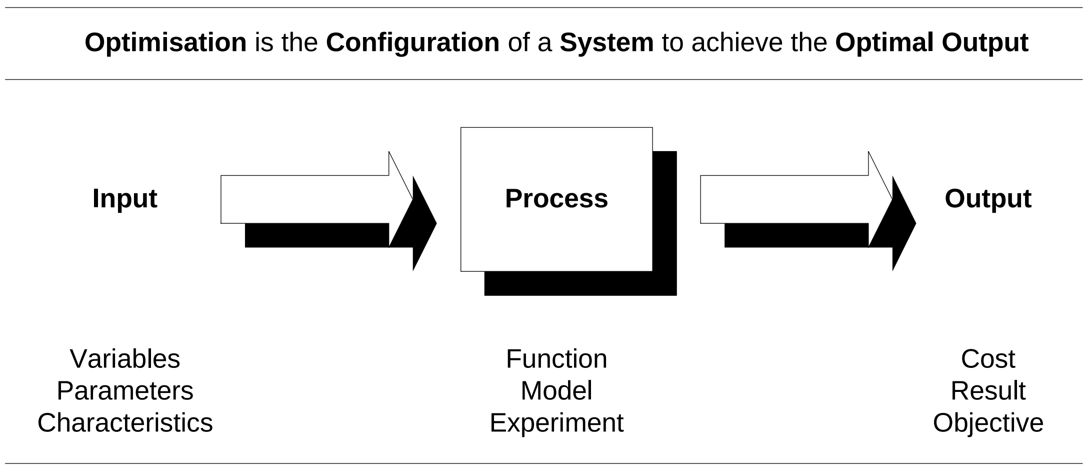


Figure 2-1 Optimisation Model

Another way to interpret the process of optimisation is in terms of searching a function's cost surface for the optimal result – in this manner, peaks or troughs in the cost surface represent the optimal result (Haupt & Haupt 2004, sec. 1.1.1).

Importantly, the task of optimisation seeks to achieve the best result possible for a given system. However, dependent upon the context or environment, the optimal result could be represented by either a maximum or a minimum result. In most contexts – especially the case for Genetic Algorithms – optimisation to find a maximum output is often referred to as maximising a system's *fitness*, whereas optimisation to find a minimum output is often referred to as minimising a system's *cost*. Thus, fitness is the negative of cost (Haupt & Haupt 2004, sec. 1.1.1). However, this project uses a more generic definition whereby fitness is simply the optimal (minimum or maximum) cost value. For ***Argo***, fitness is evaluated in terms of minimising the cost function.

Regardless of optimising a system's fitness or cost, a common challenge is finding the global minima/maxima vice any number of local minima/maxima. This is more easily visualised using the concept of a cost surface for which there may exist any number of smaller peaks and troughs.

2.3 Root Finding

Mathematically the process of root finding is similar to the process of optimisation. Root finding searches for the zeros of a function whereas optimisation searches for the zeros of a function's derivatives (Haupt & Haupt 2004, sec. 1.1.2).

Root finding does not suffer from the problem of calculating local minima/maxima, as any root is as good as another – it drives the function to zero.

Unfortunately, although root finding is mathematically well understood, in practice, most real world systems are difficult to model and solve for the roots, especially for non-linear, multi-variable, time variant systems.

2.4 Categories of Optimisation

This paper has chosen to adopt the six categories of optimisation presented by Haupt and Haupt (sec. 1.1.3). Those categories being:

a) Function / Trial & Error,

b) Single Variable / Multiple Variable,

c) Static / Dynamic,

d) Discrete / Continuous,

e) Constrained / Unconstrained, and

f) Minimum Seeking / Random.

Optimisation by Trial & Error simply adjusts the inputs and observes the outputs. Changes to inputs are made based on these outputs. No understanding of the process is applied to the problem when adjusting the inputs. Whereas optimisation by Function sets the inputs and uses an understanding of the process in order to identify the best output.

Multiple variable systems are more complex than single variable systems and are more difficult to model and solve mathematically. The number of variables can be used to express the number of dimensions within the system, for example, the number of dimensions to a cost surface.

Dynamic systems are systems for which the output is a function of time (Haupt & Haupt 2004, sec. 1.1.3) – static systems are time invariant.

System variables can be classified as either discrete or continuous. Continuous variables can take an infinite number of values; whereas discrete variables can only be assigned a finite number of possible values. A common approach to optimising continuous systems is to first discretise the system and then attempt to optimise using digital processes.

Constrained systems are systems for which variables can only take values within set limits. Variables in unconstrained systems have no such limits applied. Mathematical optimisation works best on unconstrained systems.

Minimum seeking optimisation methods use a single set of inputs in order to normally numerically find the optimal outputs. Such methods are challenged by the problem of local minima/maxima. Unlike minimum seeking optimisation methods, random methods use probabilistic calculations to find the variable sets on which to perform optimisation, thus finding local minima/maxima is not as problematic. Typically, minimum seeking methods are computationally faster than random methods.

2.5 Natural Optimisation Methods

As eluded to when discussing root finding (refer Section 2.3), most classical optimisation methods can be described as minimum-seeking algorithms searching the cost surface for minimum cost and hence suffer from the challenge of local minima. Such classical methods are often calculus based and solved numerically.

More recently, natural optimisation methods have been developed in order to address the inherent limitations of calculus-based optimisation. Haupt and Haupt (2004, sec. 1.1.3) provide five examples of natural optimisation methods including:

a) Genetic Algorithms,

b) Simulated Annealing,

c) Particle Swarm Optimisation,

d) Ant Colony Optimisation, and

e) Evolutionary Optimisation.

Such natural optimisation methods attempt to model a real-world process based on a system displayed in nature. This is done because it has been observed that nature is amazingly adept at optimising many of its natural systems.

These natural methods provide an intelligent search of the solution space using statistical methods and hence do not require finding the cost function’s derivatives; thus natural methods can handle systems with multiple, non-continuous and discrete variables (Haupt & Haupt 2004, sec. 1.3).

2.6 Summary

In summary, this chapter has reviewed some general optimisation concepts and presented a generic optimisation model – the configuration of a set of inputs to a system’s process in order to find the optimal output.

The chapter briefly also outlined some common natural optimisation techniques, including Genetic Algorithms. Such natural optimisation methods attempt to model a real-world process based on a system displayed in nature. These natural methods provide an intelligent search of the solution space using statistical methods. It is the use of statistical methods vice analytical methods that often make the use of natural methods more successful than calculus based methods for systems with multiple, non-continuous and discrete variables.

Chapter 3

Digital Controllers

3.1 Introduction

> Engineering is concerned with understanding and controlling the materials and forces of nature for the benefit of humankind. Control system engineers are concerned with understanding and controlling segments of their environment, often called *systems*, in order to provide useful economic products for society (Dorf 1992, sec. 2, p. 2).

Ignoring the economic products, this project is fundamentally concerned with the control of inputs to systems that maintain cause-effect relationships to the outputs (Dorf 1992, sec. 2, p. 2); whereby this control is based on linear system theory.

In practice, digital controllers are used to control real-world devices and processes. Leis (2003, Course Overview) presents the following examples of practical digital controller examples:

a) **Aviation –** Flight Control Systems for controlling flight control surfaces

b) **Robotics –** Motion

c) **Automotive** – Antilock Braking Systems

d) **Industrial –** temperature control systems in manufacturing

In reviewing control systems, this chapter will first present an overview of both analogue and a computer controlled systems. After providing an overview of each type of control system, a basic algorithm for any control system will be presented. Once the control system has been introduced, the controller itself along with its ideal characteristics will be discussed. Finally, the PID Controller will be defined and methods for tuning the controller's parameters outlined.

3.2 Control System Overview

3.2.1 Analogue Controller

In order to understand computer controlled systems, it is first important to briefly review the analogue controller. The analogue control system is generally comprised of a summing junction, a controller $G_c(s)$, plant $G_p(s)$, and feedback transmittance $H(s)$. The analogue controller is generally modelled in the time domain (t) with transfer functions using the S-Domain. A general form of an analogue control system is shown in Figure 3-1.

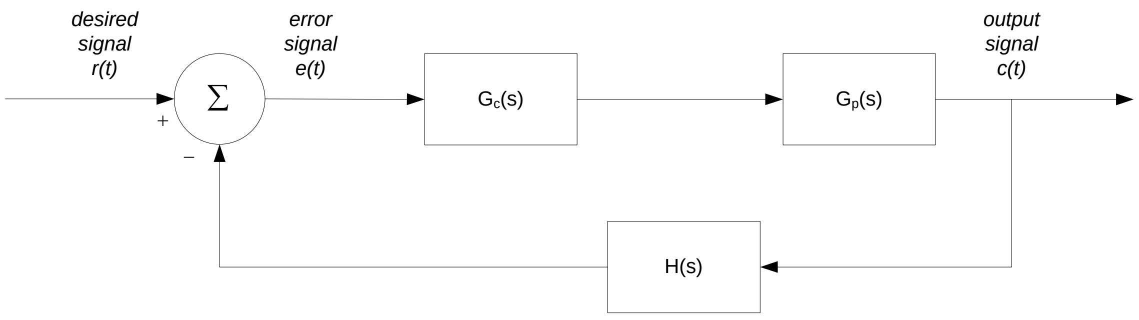


Figure 3-1 Analogue Control System (based on (ELE3105 2007, mod. 1, fig. 1.1))

Dorf (1992, sec. 1.1, p. 2) notes that feedback systems (closed-loop systems) provide a measure of the output signal back with the desired signal in order to control the system; thereby enabling the control system to drive the controller to eliminate error in the desired output signal. The feedback signal is often amplified during measurement - $H(s)$.

From the control loop shown in Figure 3-1, the University of Southern Queensland's Computer Controlled Systems' Study Guide (2007, mod. 1) suggests that the basic sequence of events for any analogue controlled system can then be described as follows:

1. Generate the desired output signal $r(t)$ at time t
2. Measure the actual output signal $c(t)$

3. Calculate the signal error $e(t) = r(t) - c(t)$
4. Apply the control algorithm to generate control signal $m(t)$
5. Output the control signal to plant input
6. Repeat step 1

3.2.2 Computer Controller

The computer controller is similar to the analogue controller, however, as the signals are now digitised, a number of other devices must be considered including digital-to-analogue converters, analogue-to-digital converters and digital samplers. Also, as the signals in a computer controlled system are discretised, the computer controlled system is generally modelled in the discrete time domain (k) with transfer functions using the Z-Domain. A general form of a computer controlled system is shown in Figure 3-2.

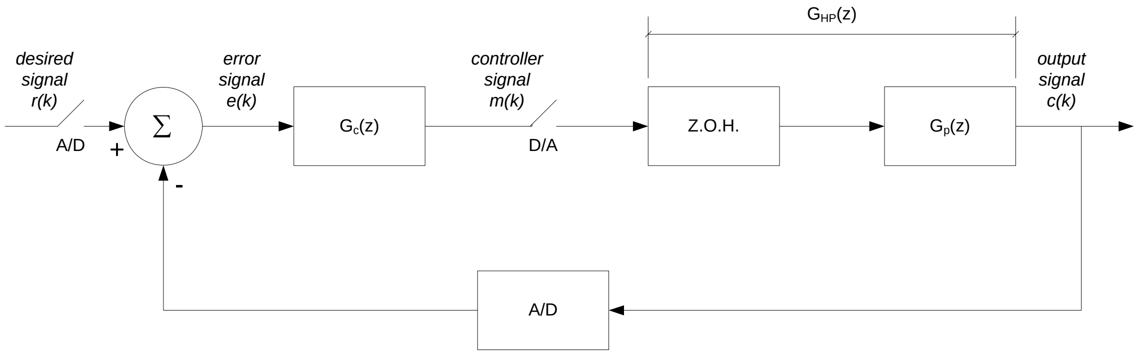


Figure 3-2 Computer Controlled System (based on (ELE3l05 2007, mod. 5, fig. 5.1))

From the digital control loop shown in Figure 3-2, the basic sequence of events for any computer controlled system can then be described in a similar manner to analogue systems in the preceding section (refer Section 3.2.1):

1. Generate the desired output $r(k)$ for sample k
2. Measure the actual output $c(k)$
3. Calculate the error $e(k) = r(k) - c(k)$
4. Apply the control algorithm to generate control signal $m(k)$
5. Output the control signal to plant input

6. Repeat step 1

3.2.3 Controllers

Now that the control system has been explained, the controller itself can be discussed. Generically, Leis (2003, mod. 4) suggests that an ideal Controller has three key characteristics:

a) Fast response,

b) Minimal overshoot, and

c) No steady-state error.

Leis (2003, mod. 4) further suggests that these characteristics can be found by the combination of the three base Controllers:

a) **Proportional Controller –** control $\propto$ error:

   i. Increase response speed;

   ii. Decrease steady-state error; and

   iii. Decrease system damping.

b) **Integral Controller –** control $\propto$ accumulated error:

   i. Accumulates while there is an error;

   ii. Forces steady-state to zero; and

   iii. Decreases stability.

c) **Derivative Controller –** control $\propto$ rate-of-change-of-error:

   i. Brakes the response; and

   ii. Makes the response more sluggish.

The three basic controller types – proportional, derivative and integral – can be practically combined to forma PID Controller. Figure 3-3 shows how a PID Controller is used within the standard digital control loop.

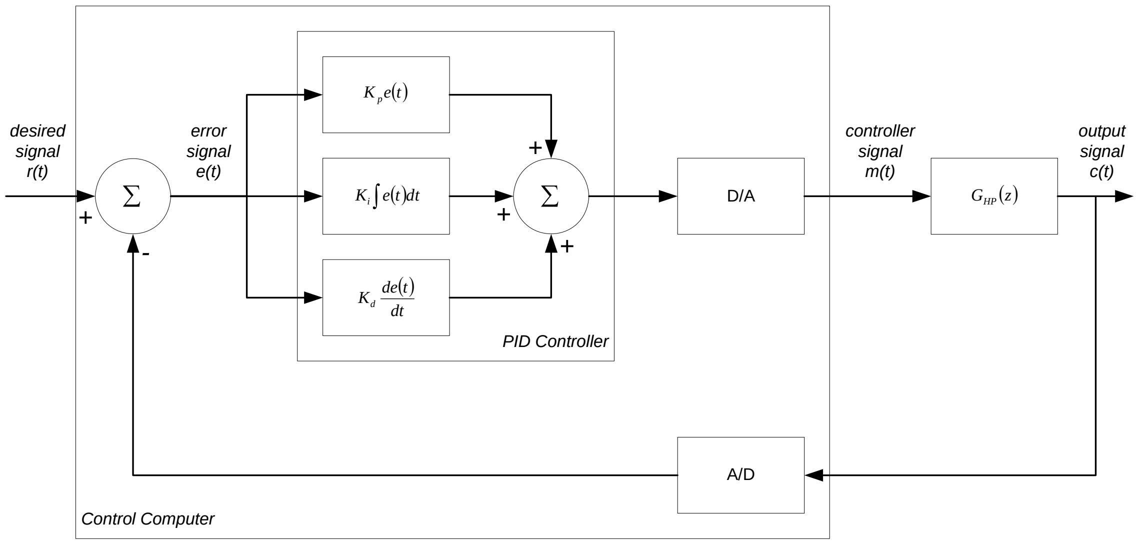


Figure 3-3 Digital Control Loop (based on (Leis 2003, mod. 1; mod.4))

Taking the standard digital control loop, the controller's signal can then be mathematically modelled as:

$$m(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt}$$

EQN 3–1

Where the constants:

$K_p = K\left(1 + \frac{T_d}{T}\right)$, and
K is gain constant
T_d is derivative time
T is sample time

EQN 3–2

$K_i = -K\left(1 + \frac{2T_d}{T} - \frac{T}{T_i}\right)$, and
K is gain constant
T_d is derivative time
T_i is reset time of the integrator
T is sample time

EQN 3–3

$K_d = \frac{KT_d}{T}$, and

K is gain constant

T_d is derivative time

T is sample time

EQN 3–4

It can be shown that this function can be transformed to:

$$M(z) = \frac{\left(q_0 + q_1 z^{-1} + q_2 z^{-2}\right)E(z)}{\left(1 - z^{-1}\right)} \qquad \text{EQN 3–5}$$

Which can then be represented as a difference equation of:

$$m_n = q_0 e_n + q_1 e_{n-1} + q_2 e_{n-2} + m_{n-1}$$

EQN 3–6

Where the constants:

$$q_0 = K\left(1 + \frac{T_d}{T}\right)$$

EQN 3–7

$$q_1 = -K\left(1 + \frac{2T_d}{T} - \frac{T}{T_i}\right)$$

EQN 3–8

$$q_2 = \frac{KT_d}{T}$$

EQN 3–9

Ultimately it is these three constants — q_0, q_1 and q_2 — that will be modelled as Genes within the Genetic Algorithm optimisation presented within this paper using ***Argo***.

3.3 Tuning

In order to achieve the desired performance of the controller the three PID parameters must be tuned. University of Southern Queensland's Computer Controlled Systems Study Guide (2007, mod. 4.2) suggests that tuning can be performed using a number of methods including:

a) Trial and error,

b) Experimental results,

c) Heuristics (e.g. Ziegler-Nichols),

d) Analytical analysis (e.g. Steepest decent optimisation), or

e) Natural algorithms (e.g. Genetic Algorithm).

This project is fundamentally concerned with providing a Genetic Algorithm from which to optimise the three PID parameters for a given control system.

The University of Southern Queensland's Computer Controlled Systems Study Guide (2007, mod. 4) also explains that tuning can be either fixed or adaptive. Fixed tuning selects the controller parameters upon start of the control system, and they remain *as-set* whilst the control system is in operation. Adaptive tuning seeks to change the parameters during operation of the control system in order to provide optimal control performance by addressing any changes to the control system during operation (including environmental changes impacting the system).

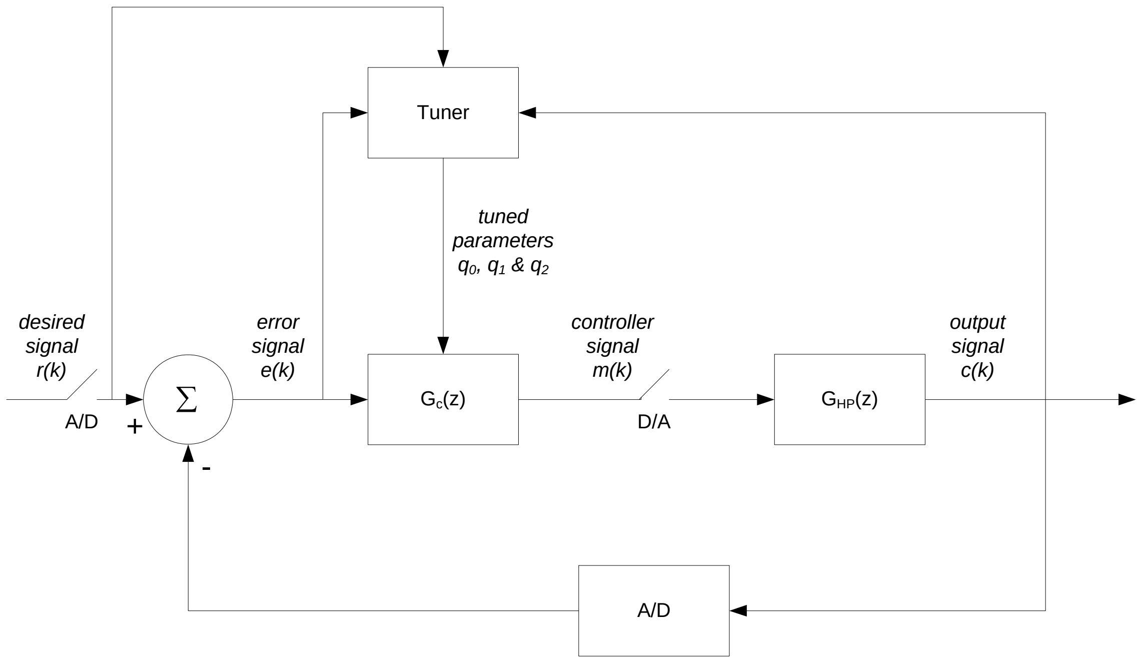


Figure 3-4 Adaptive Digital Control System

This project will provide a Genetic Algorithm for optimisation of a PID Controller using fixed tuning only. However, the project will endeavour to explain how it could be extended as the focus of further work to the project.

3.4 Summary

In summary, this chapter presented an overview of control systems, providing models for both analogue and computer control systems. The chapter then discussed the ideal characteristics for any controller, and provided a mathematical model for a controller that could achieve these requirements – the PID Controller. Finally, the chapter briefly discussed methods of how to tune the parameters of the PID Controller, including:

a) Trial and error,

b) Experimental results,

c) Heuristics (e.g. Ziegler-Nichols),

d) Analytical analysis (e.g. Steepest decent optimisation), or

e) Natural algorithms (e.g. Genetic Algorithm).

This project will use a Genetic Algorithm in order to provide a fixed tuning solution to a control system.

Chapter 4

Genetic Algorithms

4.1 Introduction

Before the MATLAB® program ***Argo*** can be explained and its interface with a PID Controller demonstrated, a sound understanding of Biological Genetic Algorithms must first be gained. To achieve this goal, this Chapter will present the general principles of Biological Genetic Algorithms including the fundamental concept of Natural Selection. Once the concept of Natural Selection has been presented, this Chapter will then explain the basic process that any Genetic Algorithm would follow if applied to a real-world problem.

In order to understand the capabilities and limitations of applying a Genetic Algorithm to an engineering optimisation problem, the Chapter concludes with a brief discussion on the main advantages and disadvantages associated with Genetic Algorithms.

4.2 Biological Genetic Optimisation

Genetic Algorithms are simply a series of steps for solving a problem whereby the problem-solving method uses genetics as the basis of its model (Chambers 2005, preface). Genetics is the branch of biology that studies how parent organisms transfer their cellular characteristics to children. Genetic Algorithms attempt to model the concept of Natural Selection within genetics, whereby Chambers (2005, preface) explains that

"… organisms most suited for their environment tend to live long enough to reproduce, whereas less-suited organisms often die before producing young or produce fewer and/or weaker young".

Implicit in the concept of Natural Selection is the idea that the stronger organisms live long enough to reproduce often and pass their genetic traits on to their offspring, whereas because the weaker organisms do not produce as many offspring their genetic traits are not as prolific within the population.

At the cellular level a gene is the basic unit of heredity (Haupt & Haupt, 2004, sec. 1.4). The gene contains information that describes a specific trait of an organism. Multiple genes are combined to form a chromosome – the sequence of genes within the chromosome is often referred to as the organism's genetic code (Haupt & Haupt 2004, sec. 1.4). One common implementation for a Genetic Algorithm is to code chromosomes and genes as a string of bits. Individual bits in the encoded string are analogous to the genetic concept of an allele (Whitley n.d., p. 16).

For a Genetic Algorithm to model the real world, the first important step is that of selection. Selection is the process where organisms are chosen to mate and produce offspring (Haupt & Haupt 2005, sec. 2.2.5). Natural Selection often occurs in nature by mating two parents to produce one offspring. Although Genetic Algorithms are not necessarily limited to a set number of parents, it is common to select only two parents for mating. ***Argo*** requires the selection of two parents for mating which in turn produces two offspring.

Once two offspring are selected mating occurs. Mating is the process that mimics sexual recombination of cells. Genetic Algorithms perform mating by a process of crossing chromosomes. Crossing is a process whereby two parent cells divide and then arrange themselves such that they recombine to form offspring that have part of their chromosome provided by both parents. Crossing mimics the genetic process of meiosis that results in cell division (Haupt & Haupt 2004, sec. 1.4) – reproductive cells divide at a random point along the chromosome known as the kinetochore (Haupt & Haupt 2004, sec. 1.4).

A rare but important part of mating is mutation. Chambers (1995, p. 48) explains that mutation "is the process of randomly disturbing genetic information". Mutation manifests itself by randomly altering a gene (more specifically an allele) within the chromosome. Similar to nature, Genetic Algorithms seldomly apply the process of mutation. Natural Selection leads to the maintenance of a strong genetic population through heredity of strong genes. Mutation counters this, and hence is used to reintroduce alleles/genes that may have been lost in the population after selection and crossing but which actually make the chromosome stronger in terms of its environment. Genetic Algorithms apply mutation in order to avoid prematurely converging to a sub-optimum genetic solution.

4.3 Genetic Algorithms

Obviously nature applies the processes of selection, mating, crossing, mutation and reproduction continuously whilst ever the species continues to survive. However, Genetic Algorithms are an optimisation and search technique based on the principles of genetics and natural selection in order to maximise genetic fitness (Haupt & Haupt 2004, sec. 1.5). Often Genetic Algorithms encode the parameters of a real-world problem and then attempt to maximise an associated fitness function (Whitley n.d., p. 1). Thus, whereas nature applies the process of natural selection continuously, Genetic Algorithms apply the process iteratively until a set of encoded parameters is found that maximises the modelling function. Hence Genetic Algorithms are often used as a function optimising technique.

The key difference between Genetic Algorithms and analytical optimisation is that in effect Genetic Algorithms are a population-based model that searches the fitness space to find the optimum parameters (Whitley n.d., p. 1). Whereas analytical optimisation attempts to mathematically model the process and optimise using either calculus or numerical techniques.

4.4 Advantages and Disadvantages

As already eluded to, Genetic Algorithms have numerous inherent advantages over classical numerical optimisation techniques. Haupt & Haupt (2005, sec. 1.5) attest that some of the advantages of Genetic Algorithms are that they:

a) Can handle discrete and continuous variables;

b) Don’t require the calculation of function derivates (not calculus based);

c) Are suited to parallel computing (still the current means from which personal computers are attempting to gain significant increases in processing power);

d) Can provide a list of optimal variables;

e) Can handle complex cost surfaces (local minima/maxima do not falt the method); and

f) Can handle large numbers of variables.

However, despite the many advantages over classical analytical optimisation techniques, the Genetic Algorithms own process of searching a large solution space results in a significant disadvantage. That disadvantage being a high computational cost associated with processing and searching a large solution space. Such a computational cost is normally manifested by a slow computational process and a high demand for memory. Hence, classical analytical optimisation techniques still remain the best for complex analytical functions with few variables.

4.5 Genetic Algorithm Process

4.5.1 Problem Definition and Encoding

The generic Genetic Algorithm process is shown in Figure 4-1 Biological Genetic Algorithm Process Flow. The process commences with the definition of the problem and the encoding of chromosomes from which to apply the Genetic Algorithm process digitally.

This step is also important as the convergence criteria must be defined. The convergence criterion defines the situations under which the Generic Algorithm will be deemed completed.

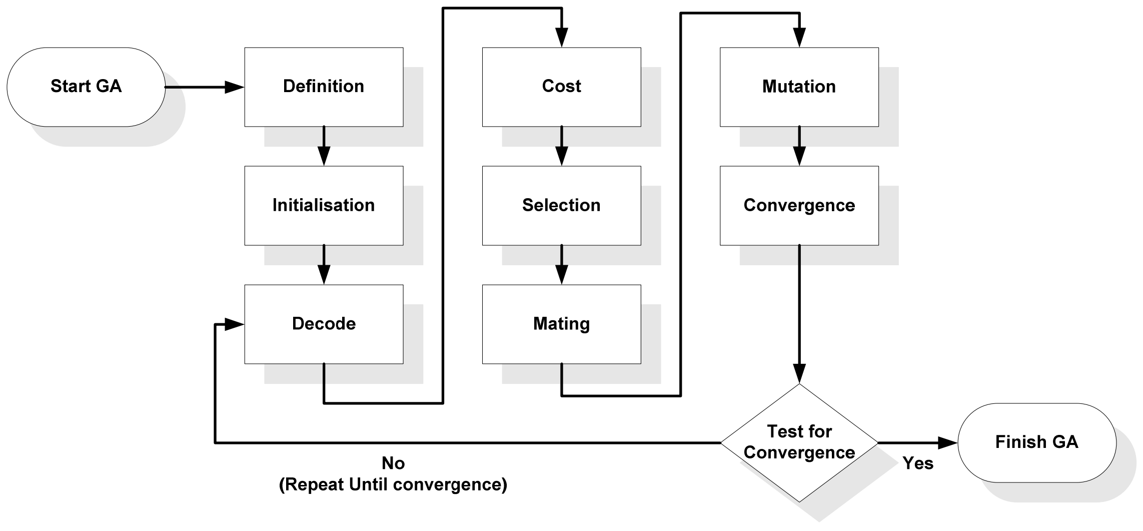


Figure 4-1 Biological Genetic Algorithm Process Flow

4.5.2 Initialisation

Once coding of chromosomes is defined, the starting population of chromosomes within the search space can be initialised. The process of initialisation can be tailored dependent upon the problem and how the system operates in practise. For example, initialisation could be achieved by setting all chromosomes to be the same sequence with the chromosome value being arbitrarily chosen, selected at random or specifically nominated. Regardless, this method is not normally recommended as the Genetic Algorithms strength lies in its ability to provide and search a diverse solution space, and this method slows the diversification of the population – and hence slows the optimisation process in general. However, the speed of the optimisation problem could be increased if the arbitrary value representing the chromosome was known to be near the optimal value.

Nevertheless, if the optimal value for a chromosome is unknown, the best initialisation method of the population is to randomly assign values to all chromosomes within appropriate limits for the system variables that the chromosomes are modelling.

4.5.3 Decoding

The next step in the process is to decode the chromosome’s value to enable calculation of the cost. Obviously this step depends upon the coding scheme used. For ***Argo***, a binary code was utilised.

4.5.4 Cost

Once each chromosome has been decoded, the chromosome's cost can then be calculated. The cost function is normally a mathematical function, but, it could be programmed to be derived from an experimental result or even an outcome from a game (Haupt and Haupt, 2004, sec 2.2.1). Referring to the generic optimisation model presented in Chapter 2 (refer Section 2.2), the cost is the system's output derived from a set of inputs. Further, the cost can also be considered to be the difference between the actual output value for the given set of inputs and the optimal output (Haupt and Haupt, 2004, sec 2.2.1).

As previously discussed in Chapter 1 (refer Section 1.4.1), Whitley (n.d.) prescribes that there is a difference between the evaluation and fitness functions within the Genetic Algorithm. The evaluation function determines the cost independent of evaluation of other chromosome's cost. Whereas the fitness function determines the cost of the chromosome relative to all other chromosome's within the population. Practically this could take the form of a simple sorting algorithm.

4.5.5 Selection

Once each chromosome's cost has been calculated, the Genetic Algorithm can use this data to determine which chromosomes should be selected to mate. The selection function is generally the aspect that contains the most differences between Genetic Algorithms and arguably has the greatest impact upon the Genetic Algorithm's eventual success.

A number of selection methods have been proposed including:

a) Ranked pairing,

b) Random pairing,

c) Weighted random pairing, and

d) Tournament Selection.

Ranked pairing is a simple selection method whereby two adjacent chromosomes are selected from a rank sorted list, whereby ranking is based on cost. Extensions to this method include Fit-Fit and Fit-Weak selection (Chambers, 1995, sec. 1.14.1.3 and

1.14.1.4). Fit-Fit (pairing with next fittest chromosome) is highly conservative compared to Fit-Weak (pairing with next weakest chromosome) which is highly disruptive of the genetic information. Obviously ranked pairing does not follow nature's model, however, it is simple to program (Haupt and Haupt, 2004, sec. 2.2.5). This method also has the computational penalty of requiring a sort of each population.

Random pairing is also a simple selection method whereby two chromosomes are selected at random from the population, regardless of ranking. This method has some similarities with nature, and, has the added advantage of not requiring the computational cost of sorting each population.

Weighted Random selection, also known as Roulette Wheel selection, is one of the most common selection methods used in practical Genetic Algorithms. Roulette Wheel selection is based on its namesake whereby each slot in the wheel is weighted in proportion to its fitness. Hence, selection is more likely for fitter chromosomes. A random number is used to determine which chromosome on the Roulette Wheel is selected. Weighted Random selection can be refined by using either the ranking or the cost to calculate the probability of selection (Haupt and Haupt, 2004, sec. 2.2.5). This method has similarities with nature, as it attempts to provide a model that mimics the concept of Natural Selection. However, this method is computationally expensive. Also, Chambers (1995, Sec 1.14.1.1) notes that this method is "only a moderately strong selection technique, since fit individuals are not guaranteed to be selected for, but have a somewhat greater chance". Chambers (1995, Sec 1.14.1.1) also warns for practical application that it is essential not to sort the population, as this will dramatically bias the selection.

Haupt and Haupt (2004, sec. 2.2.5) suggests that Tournament Selection is perhaps the method that most closely mimics nature. This method identifies a pool of candidate chromosomes at random and then selects the fittest candidate as the first successful parent. This method has the added advantage of not requiring the computational cost of sorting each population.

4.5.6 Mating

The next step in the Genetic Algorithm is to mate the parent chromosomes to form an offspring. Typically, mating uses the process of crossover whereby a crossover point (kinetochore) is chosen and the genetic codes of each parent are then swapped around that point (Haupt and Haupt, 2004, Sec 2.2.6). Multiple crossover points can be used, however, the greater the number of crossover points used the greater the disruption of the genetic information in the population. Normally the crossover point itself is chosen at random from the genetic code (Haupt and Haupt, 2004, Sec 2.2.6).

4.5.7 Mutation

Mutation is a change in a gene's characteristics. The change occurs randomly in order to re-introduce/introduce genetic traits not found in the population. This process is key to a Genetic Algorithm's ability to eventually converge on the global optimal solution – mutation helps prematurely converging on a local minima/maxima.

However, mutation by concept is disruptive. Therefore, the mutation rate for most Genetic Algorithms is very low, often in the order of 1% or lower. Also, it is normal practice for most Genetic Algorithms not to allow mutation on the current fittest chromosome.

Practically, mutation on a binary encoded Genetic Algorithm is simply a change in an allele's value from a '1' to a '0' or vice versa. Also, although most Genetic Algorithms only allow a single mutation to occur on any given gene, they can be tailored to allow any number of mutations. However, it is important to realise that the more mutations possible, the more disruptive the Genetic Algorithm. In practice, this may be a useful mechanism to help solve optimisation problems for which it is known *a priori* that there are numerous local maxima/minima.

4.5.8 Convergence

The final step in any Genetic Algorithm is the test for convergence. Nature evolves making the species stronger by adapting to its environment whilst ever the species itself exists; whereas the purpose of a Genetic Algorithm is to eventually converge on a single solution to an optimisation problem. Thus, as part of the initial problem definition phase convergence criteria are normally defined, including tolerances. Thus, at the completion of selection, mating, and mutation of the current population a convergence test is performed. If the Genetic Algorithm is deemed to have successfully converged on the optimal solution then the Genetic Algorithm will terminate. If not, the Genetic Algorithm will continue again using the end-state of the current population as the stating population for the next generation.

It is also normal practice for all Genetic Algorithms to hardcode exit criteria in addition to any convergence criteria. Typically, such exit criteria includes a limit to the total number of generations allowed on any run of a Genetic Algorithm. As Genetic Algorithms rely upon random number selection for their operation, some Genetic Algorithms may take many more generations to converge upon the optimal solution, if at all.

4.6 Summary

In summary, this Chapter discussed the basic concept of biological genetic optimization. It was noted that Genetic Algorithms attempt to model the concept of Natural Selection in order to solve an optimization problem. Natural Selection is a concept whereby the stronger organisms live to reproduce and pass their genetic traits to their offspring; whereas because the weaker organisms do not produce as many offspring their genetic traits are not as prolific within the population. Obviously nature applies the processes of selection, mating, crossing, mutation and reproduction continuously whilst ever the species continues to survive. However, Genetic Algorithms are an optimisation and search technique based on the principles of genetics and Natural Selection in order to maximise genetic fitness (Haupt & Haupt 2004, sec. 1.5).

The chapter also briefly considered the advantages and disadvantages of using a Genetic Algorithm as an optimization method over classical analytical optimization techniques. Haupt & Haupt (2005, sec. 1.5) attest that some of the advantages of Genetic Algorithms are that they:

a) Can handle discrete and continuous variables;

b) Don’t require the calculation of function derivates (are not calculus based);

c) Are suited to parallel computing (still the current means from which personal computers are attempting to gain significant increases in processing power);

d) Can provide a list of optimal variables;

e) Can handle complex cost surfaces (local minima/maxima do not falt the method); and

f) Can handle large numbers of variables.

However, despite the numerous advantages that Genetic Algorithms provide, perhaps their greatest disadvantage results from the high computational cost associated with processing and searching a large solution space.

Finally, the chapter concluded with a detailed overview of a basic Genetic Algorithm, specifically the iterative process of:

Step 1. Problem definition and encoding

Step 2. Population initialization

Step 3. Decoding chromosome values

Step 4. Calculation of chromosome cost/fitness

Step 5. Selection of parent chromosomes

Step 6. Mating of parents to generate offspring – new chromosomes

Step 7. Mutation of the genetic code

Step 8. Convergence against pre-determined criteria for identifying optimal solution

Step 9. Repeat step 3 if not converged.

Chapter 5

Argo

5.1 Introduction

Thus far the dissertation has discussed optimisation in general, the concept of Natural Selection at length, the common Genetic Algorithm itself, and a brief overview of digital controllers. Thus the foundation has been laid on which to present the projects Genetic Algorithm coded in MATLAB^R – ***Argo***.

This chapter will first define the problem and the method of encoding for the binary Genetic Algorithm. Then the chapter will explain the Genetic Algorithm used and follow-on to cover the functions that perform initialisation, cost evaluation, selection, mating, mutation and convergence.

5.2 Definition

In relation to the optimisation model presented in Chapter 1 (refer Figure 2-1), the genes that comprise the chromosome represent the *inputs*. The fitness/cost function represents the *process* and the chromosome fitness values represent the *outputs*.

Generically, a chromosome is represented as:

$$\text{Chromosome } = [g_1, g_2, g_3 \ldots g_n]$$

EQN 5–1

Where,

$$\mathrm{g_n} = \text{ gene number } \mathrm{n}$$

EQN 5–2

And,

$$\begin{aligned} Fitness &= f(Chromosone) \\ &= f(g_1, g_{2,} g_3 \ldots g_n) \end{aligned}$$

EQN 5–3

For ***Argo***, there are three genes comprising a single chromosome. The genes represent the parameters within the generic PID Controller function (refer Section 3.1):

$$m_n = q_0 e_n + q_1 e_{n-1} + q_2 e_{n-2} + m_{n-1}$$

EQN 5–4

Where the constants represent the three genes,

$$q_0 = g_1$$

EQN 5–5

$$q_1 = g_2$$

EQN 5–6

$$q_2 = g_3$$

EQN 5–7

5.3 Encoding

Before each function within ***Argo*** can be described, the method of encoding chromosomes must first be presented. ***Argo*** encodes three PID parameters – q_0, q_1, q_2 – as three genes in a chromosome. In order to meet the project specification, each gene is 30 bits. This results in the following maximum operating limits for each parameter:

$$+1073741823 < q_n < -1073741823$$

EQN 5–8

However, for any given value of q, a division by 10,000 will be performed to achieve the decimal accuracy required by the project. Thus the practical operating limits for each parameter are:

$$+10737.41823 < q_n < -10737.41823$$

EQN 5–9

In addition, a single sign bit will be appended at the front of each gene. Hence, each gene is constructed as:

$$floats\,[+/-]\,xxxxx\,[.]\,xxxxx$$

EQN 5–10

Where *x* can be either a:

a) 1 (representing a positive number), or

b) 0 (representing a negative number).

Note, the decimal point is not actually coded – division by a 10,000 performed procedurally. Thus each gene is actually 31 bits:

$$sign\ (1\ bit) + magnitude\ (30\ bits) = 31\ bits$$

EQN 5–11

And therefore a complete chromosome is 93 bits:

$$3\ x\ genes\ =\ 93\ bits$$

EQN 5–12

Changing the project specification or method of encoding would require a significant redesign of ***Argo***.

The `ParseBits` function is used to decode a chromosome's binary code.

5.4 *Argo* Genetic Algorithm

The C++® source code provided online by Buckland (2005b) was used to provide a conceptual template for how the main routine could be structured within any practical Genetic Algorithm (akin to the process illustrated by Figure 4-1)**.** This conceptual framework was used as a starting point and subsequently extensively modified and built upon to meet the specific requirements of ***Argo*** itself.

The high-level Genetic Algorithm applied by ***Argo*** is outlined in Figure 5-1.

```
%   set Constants
%      GENE_LENGTH => 30
%      CHROMO_LENGTH => 93
%      POP_SIZE => 50
%      FALSE => 0
%      TRUE => 1
%
%   set Parameters
%      SELECTION => 'Tournament'
%      NUMBER_OF_CROSSOVER_POINTS => 1
%      DISTRIBUTION => 'uniform'
%      MUTATION_RATE => 0.001
%      MUTATIONS_PER_CHROMOSOME => 1
%      MAX_GENERATIONS => 10000
%      NKEEP => 0.7
%      SAME_MIN => 20
%
```

```
%   initialise all variables
%
%   initialise population using function [InitialisePopulation]
%
%   calculate fitness scores for population using function
%      [CalcFitness]
%   keep the best chromosome Old_Best_Chromo

%   while Generations < than MAX_GENERATIONS AND Stop == FALSE
%
%      create Intermediate_Population
%
%      loop POP_SIZE – inter_pop_size times
%         select two parents using function
%            [TournamentSelection]
%         crossover the parents to form offspring using
%            function [Crossover]
%         mutate the offspring using function [Mutate]
%
%      replace the parents with the new offspring
%
%      keep the new best chromosome => New_Best_Chromo
%      population => Intermediate_Population
%
%      calculate fitness scores for new population using
%         function [CalcFitness]
%
%   end while loop
```

Figure 5-1 ***Argo*** Genetic Algorithm

To aid in understanding how ***Argo*** operates, the program's Data Flow Diagram is provided (refer Figure 5-2).

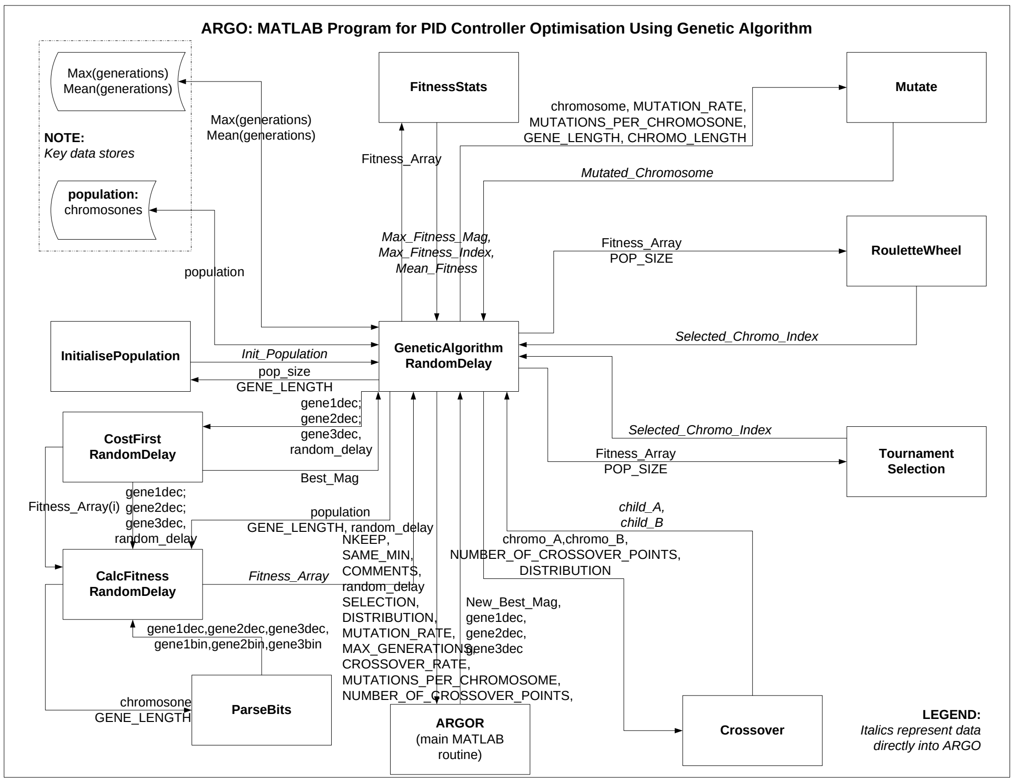


Figure 5-2 Data Flow Diagram

5.5 Initialisation

The first function within ***Argo*** is initialisation of the starting chromosome population – also referred to as the first generation. Initialisation is used to randomly assign values to all chromosomes within the population. This is done to ensure that the Genetic Algorithm starts with an initial population that has values that are spread within the input's operating limits.

The initialisation algorithm is provided in Figure 5-3.

```
%   seed initial population based on:
%       (i,j)=(rand*(10737.41823))*100000
%   ensure all numbers are rounded down
%
%   convert decimal to binary string:
%   loop for each three components (genes) of the seeded matrix
%       convert using dec2str
%       convert using num2str
%       concatenate string of three genes into a chromosome
%       concatenate chromosomes to form the population
%   end
%
%   randomly set each sign bit of each gene (bits 1,32,63)
```

Figure 5-3 Initialization Function Algorithm

5.6 Cost

This project uses three cost functions in order to test the system across a broad range of applications:

a) `CostFirst` – models a first order control system,

b) `CostSecond` – models a second order system, and

c) `CostFirstRandomDelay` – models the same first order control system but with a random delayed input.

Appendix C Cost Functions provides the mathematical background for calculating the respective costs including difference equations for the output and error {$c(k)$ and $e(k)$ respectively}, as well as derivation of the difference equations to support a random delayed input.

It is the error function that maps directly to derivation of cost through the application of the ITAE Performance Criterion. Whereby, the cost function, S, is calculated by:

$$S = \sum_{k=1}^{m} k|e(k)|$$

EQN 5–13

Where,

m is the maximum number of simulations, in this case set to 100; and

k is the discrete sample number from 1 to m .

Also, optimisation of the cost function, S, is performed using a simple implementation of the steepest descent method.

5.6.1 CostFirst

The `CostFirst` function is modelled on a transfer function for a first order plant $G_p(s)$, such that:

$$G_p(s) = \frac{5e^{-5Ts}}{s+5}$$

EQN 5–14

With an assigned sample interval of,

$$T = 0.1$$

EQN 5–15

The built-in MATLAB® function `fminsearch` indicated that the global minimum in terms of the error value was obtained with the parameter values:

$$q_0 = 1.2360$$
$$q_1 = -1.7336$$
$$q_3 = 0.6342$$

EQN 5–16

$$\cos t_{\min} = 101.7713$$

EQN 5–17

5.6.2 CostSecond

The `CostSecond` function is modelled on a transfer function for a second order plant $G_p(s)$, such that:

$$G_p(s) = \frac{(s+3)}{(s^2 + 2.s + 109)}$$

EQN 5–18

With an assigned sample interval of,

$$T = 0.05$$

EQN 5–19

The built-in MATLAB® function `fminsearch` indicated that the global minimum in terms of the error value was obtained by the parameter values:

$$q_0 = 878.1620$$
$$q_1 = 1127.8283$$
$$q_3 = 464.0663$$

EQN 5–20

$$\cos t_{\min} = 99.4803$$

EQN 5–21

5.6.3 CostFirstRandomDelay

The `CostFirstRandomDelay` function is modelled on the same transfer function for the first order plant $G_p(s)$, such that:

$$G_p(s) = \frac{5e^{-5Ts}}{s+5}$$

EQN 5–22

With an assigned sample interval of,

$$T = 0.1$$

EQN 5–23

The random delay input is defined by calculating a random number around a user set mean and standard deviation values (refer Appendix C Cost Functions).

5.7 Natural Selection

5.7.1 Tournament Selection

Tournament Selection in ***Argo*** is performed by using the `TournamentSelection` function. The `TournamentSelection` algorithm is outlined in Figure 5-4. ***Argo*** uses a Tournament Selection algorithm to perform selection of parent chromosomes. The `TournamentSelection` function simply selects three candidate parent chromosomes at random from the current population. The function then calculates the fitness of each chromosome. The fittest chromosome is then selected as the parent chromosome. ***Argo*** calls the `TournamentSelecton` function twice in order to select two parents – noting ***Argo*** selects two parents to produce two offspring.

```
%   randomly select candidate_parent_1_index from POP_SIZE
%   randomly select candidate_parent_2_index from POP_SIZE
%   randomly select candidate_parent_3_index from POP_SIZE
%
%   check that each candidate parent is unique
%
%   calculate fitness of candidate_parent_1_index
%   calculate fitness of candidate_parent_2_index
%   calculate fitness of candidate_parent_3_index
%
%   find minimum candidate parent
%
%   return the minimum candidate parent as the selected parent
```

Figure 5-4 Tournament Selection Algorithm

5.7.2 Roulette Wheel

Roulette Wheel selection in ***Argo*** is performed using the `RouletteWheel` function. The `RouletteWheel` algorithm is outlined in Figure 5-5. ***Argo*** uses a Roulette Wheel algorithm to perform selection of parent chromosomes. The `RoulettWheel` function assigns a probability to each chromosome based on its fitness. The function randomly picks a target value (slice point) and the population is stepped through until the target value is reached. ***Argo*** calls the `RouletteWheel` function twice in order to select two parents – noting ***Argo*** selects two parents to produce two offspring.

```
%   Calculate total fitness value using
%      Total_Fitness=sum(abs(Fitness_Array));
%
%   Assign each chromosome a prob based on fitness score using
%      Prob_Array=abs(Fitness_Array)./Total_Fitness;
%
%   Calculate the Cumulative Probability Array => Cum_Prob_Array
%
%   Check that the total probability equals 1
%
%   Randomly assign the selection point
%
%   Find the chromosome by
%      loop POP_SIZE number of times
%         test if Selection Point < Cum_Prob_Array(index)
%            set index
%         end
%      end
%
%   Test if index not set, then assign to last chromosome
%
%   Assign Selected_Chromo_Index => Index;
```

Figure 5-5 Roulette Wheel Selection Algorithm

5.8 Mating

Mating in ***Argo*** is performed using the `Crossover` function. The Crossover algorithm is outlined in Figure 5-6. The `Crossover` function has two configurable parameters. The first configurable parameter is the number of crossover points that can be set to either one or two points. The second configurable parameter is the type of distribution used to randomly select the crossover points. The distribution can be set as either a uniform or normal distribution.

```
%   randomly select crossover point using
%      abs(floor(CHROMO_LENGTH*normrnd(.5,.25))) or
%      floor(CHROMO_LENGTH*rand);
%
%   if crossover point is a sign bit (1-32-63) then
%      crossover point = crossover point + 1
%
%   get prefixB => (from 1:crossover point)
%   get postfixA => (from crossover point+1:CHROMO_LENGTH)
%   get postfixB => (from crossover point+1:CHROMO_LENGTH)
%   Crossed_chromosome_A => strcat(prefixA and postfixB)
%   Crossed_chromosome_B => strcat(prefixB and postfixA)
```

Figure 5-6 Crossover Algorithm

5.9 Mutation

Mutation in ***Argo*** is performed using the `Mutate` function. The Mutation algorithm is outlined in Figure 5-7. The `Mutate` function has two configurable parameters. The first configurable parameter is the mutation rate that can be set to any number between one and zero. The mutation rate is used to test whether a single bit (allele) will invert (mutate) or remain unchanged. The second configurable parameter is the number of mutated bits allowed in any given chromosome.

```
%   loop chromosome length number of times
%      test if index is not a sign bit (1-32-63)
%         test if rand is less than mutation rate
%            invert sign of bit
%            increment counter
%            test if counter is less than mutation number
%         end loop
%      end loop
%   end loop
```

Figure 5-7 Mutation Algorithm

5.10 Convergence

Importantly, because the optimum chromosome may never actually evolve, convergence testing is used to test for improvement in the population rather then stopping upon a singular evolutionary event. To achieve this ***Argo*** uses a bank of statistics. These statistics are updated and stored each generation in order to test against the trends of the statistics as well as the individual statistics themselves. In broad terms, the convergence testing is used to stop the Genetic Algorithm when the fitness scores in the population cease to improve. The statistics measured are:

1. **`Min_array`**: An array of the best fitness value for each generation.
2. **`Min_Fitness_Mag`**: The magnitude of the fitness value for the fittest chromosome in the population.
3. **`SameMin`**: The number of times (generations) the fittest chromosome has not changed.
4. **`SmallerMin`**: A Boolean used as flag to denote if the current population has a smaller best fitness value then the previous generation.
5. **`SmallerMean`**: A Boolean used as flag to denote if the current population has a better mean fitness value then the previous generation.

Argo can be configured to allow only a set number of generations to evolve without an improvement to the optimum chromosome. This is controlled via the parameter `SAME_MIN`.

Argo also tests for improvement in both the minimum and mean fitness values. The minimum value is used to track the fittest chromosome, whereas the mean is used to track the overall fitness of the population.

To simply testing, ***Argo*** places the fittest chromosome at the first position in the population.

The convergence algorithm used in ***Argo*** is outlined in Figure 5-8.

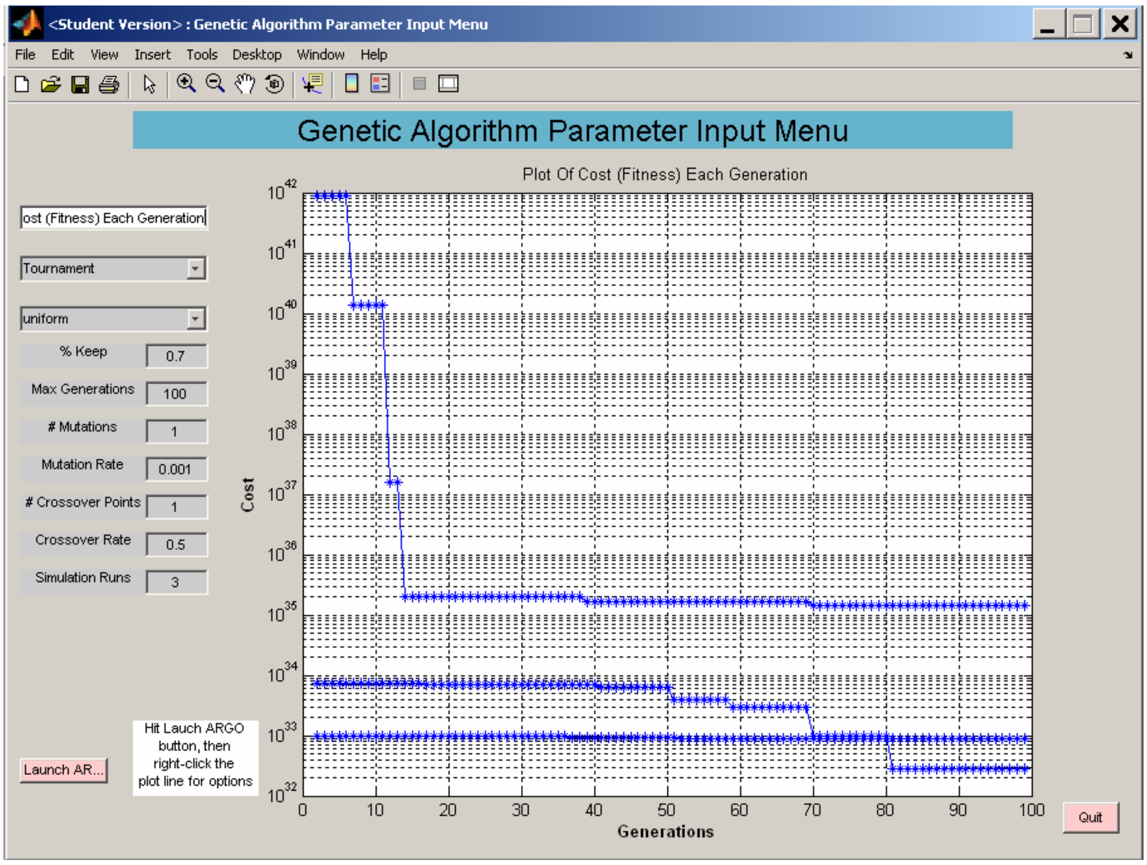
```
%   test each generation test for improving fitness
%   calculate the population’s fitness statistics

%   test if Min_Fitness_Mag equal to the minimum of Min_array
%      if yes then increment SameMin
%      else if no then reset SameMin
%
%   test if Min_Fitness_Mag is less than minimum of Min_array
%      if yes then then SmallerMin => TRUE
%      else if no then SmallerMin => FALSE
%
%   test if Mean_Fitness is less than mean of Min_array
%      if yes then SmallerMean => TRUE
%      else if no SmallerMean => FALSE
%
%   test each generation for improving fitness by
%   test if SmallerMin is TRUE
%      OR SmallerMean is TRUE
%      OR SameMin less than SAME_MIN
%         if yes continue and evolve next generation
%            (improvement in fitness)
%         if no then stop
%               (no improvement in fitness)
```

Figure 5-8 Convergence Testing

5.11 *Argo* Input And Control GUI

A basic Graphical User Interface (GUI) for input of the Genetic Algorithm's parameters and control the simulation process was developed in order to facilitate ease of operation of testing. A screen shot of the basic GUI is shown at Figure 5-9. The MATLAB® source code provided online by Land (2007) was used as a template for the GUI framework and was subsequently extensively modified and altered to meet the GUI requirements for ***Argo***.



<Student Version> : Genetic Algorithm Parameter Input Menu
File Edit View Insert Tools Desktop Window Help
Genetic Algorithm Parameter Input Menu
Plot Of Cost (Fitness) Each Generation
Cost
Generations
10^{42}
10^{41}
10^{40}
10^{39}
10^{38}
10^{37}
10^{36}
10^{35}
10^{34}
10^{33}
10^{32}
0
10
20
30
40
50
60
70
80
90
100
Launch AR...
Quit

ost (Fitness) Each Generation

Tournament

uniform

% Keep	0.7
Max Generations	100
# Mutations	1
Mutation Rate	0.001
# Crossover Points	1
Crossover Rate	0.5
Simulation Runs	3

Hit Lauch ARGO button, then right-click the plot line for options

Figure 5-9 ***Argo*** Input and Control GUI

5.12 Summary

In summary, this chapter has described the structure and operation of the Genetic Algorithm program itself – ***Argo***. The chapter also provided a brief overview of the algorithms for the key functions that performed initialisation, cost evaluation, selection, mating, mutation and convergence.

Chapter 6

Testing & Analysis Of Results

6.1 Introduction

This chapter presents the testing and analysis of test results conducted on ***Argo*** to perform verification against the project specification. The chapter is split into two sections to individually address the basic and advanced project specifications. For ease of readability, the results pertaining to the performance controlling the first order test control system with delayed inputs has been captured alongside the advanced project specifications in section 6.3.

For ease of understanding, the test plan is illustrated in Table 6-1.

Table 6-1 Test Plan

Performance Testing	**Project Specification Under Test**	**Test Phase**	**Description Of Test**	**Performance Baseline (Test Control System)**
Basic ***Argo*** Operation	Base Specifications (refer Appendix A Project Specification, Para 2)	A	Setting GA Parameters	2nd Order System
		B	Converging To Optimal PID Parameters	1st & 2nd Order Systems
		C	Number Of Generations Required For Convergence	1st & 2nd Order Systems

		D	Speed Of Convergence	2nd Order System
Advanced ***Argo*** Operation	Advanced Specifications (refer Appendix A Project Specification, Para 5 and 6)	E	Performance Using Roulette Wheel Selection	2nd Order System
		F	Simulated Control Of Rotary-Wing Model	Rotary-Wing System
		G	Performance Controlling 1st Order System With A Delayed Input	1st Order System With Delayed Input

6.2 Basic *Argo* Operation

Performance testing of basic ***Argo*** functionality was achieved by comparison against a known performance baseline. Test control systems were used as this baseline (refer Section 1.5.3 and Section 5.6). Testing used both a first order and a second order test control system in order to achieve results from a broader application.

6.2.1 First Order Test Control System

In essence, the first order test control system baseline was the optimal cost value of:

$$\mathrm{cos}\,t_{\mathrm{min}} = 101.7664$$

Resulting from the optimal PID parameter values of:

$$q_0 = 1.2360$$
$$q_1 = -1.7337$$
$$q_3 = 0.6342$$

6.2.2 Second Order Test Control System

Likewise, the second order test control system baseline was the optimal cost value of:

$\cos t_{\min} = 98.8382$

Resulting from the optimal PID parameter values of:

$q_0 = 878.1620$
$q_1 = 1127.8283$
$q_3 = 464.0663$

6.2.3 Conduct of Base Performance Testing

To analyse the base performance of the Genetic Algorithm the testing was conducted in four phases. The phases were:

a) **Phase A –** Testing Genetic Algorithm parameters to aid optimising performance (testing against second order test control system only).

b) **Phase B –** Testing Genetic Algorithm outputs for convergence to optimum PID parameter values (testing against both first and second order test control systems).

c) **Phase C –** Testing to identify the number of generations required by the Genetic Algorithm to achieve optimum results (testing against both first and second order test control systems).

d) **Phase D –** Testing to determine how quickly the Genetic Algorithm converges to optimum results to identify suitability for practical applications (testing against second order test control system only).

6.2.3.1 Base Performance Testing – Phase A

Phase A testing aims to determine which parameters are significant in optimising the performance of the Genetic Algorithm. Further, Phase A testing will also attempt to identify a set of parameter values which can be used with some confidence to achieve good performance.

Only the second order test control system was used as a baseline for ‘Base Performance Testing – Phase A’ in order to simplify test procedures. The second order system was used as it represents a more complex control system.

The parameters analysed included:

a) `TEST A1 NUMBER_OF_CROSSOVER_POINTS`: Number of crossover points,

b) `TEST A2 DISTRIBUTION`: Distribution type,

c) `TEST A3 NKEEP`: % of population kept each generation,

d) `TEST A4 CROSSOVER_RATE`: Crossover Rate, and

e) `TEST A5 MUTATION_RATE`: Mutation Rate.

6.2.3.1.1 Base Performance Testing – Test A1

The Genetic Algorithm performed reasonably well after only 100 generations. Figure 6-1 shows that there is no significant advantage to optimizing the test control system by varying either one or two crossover points.

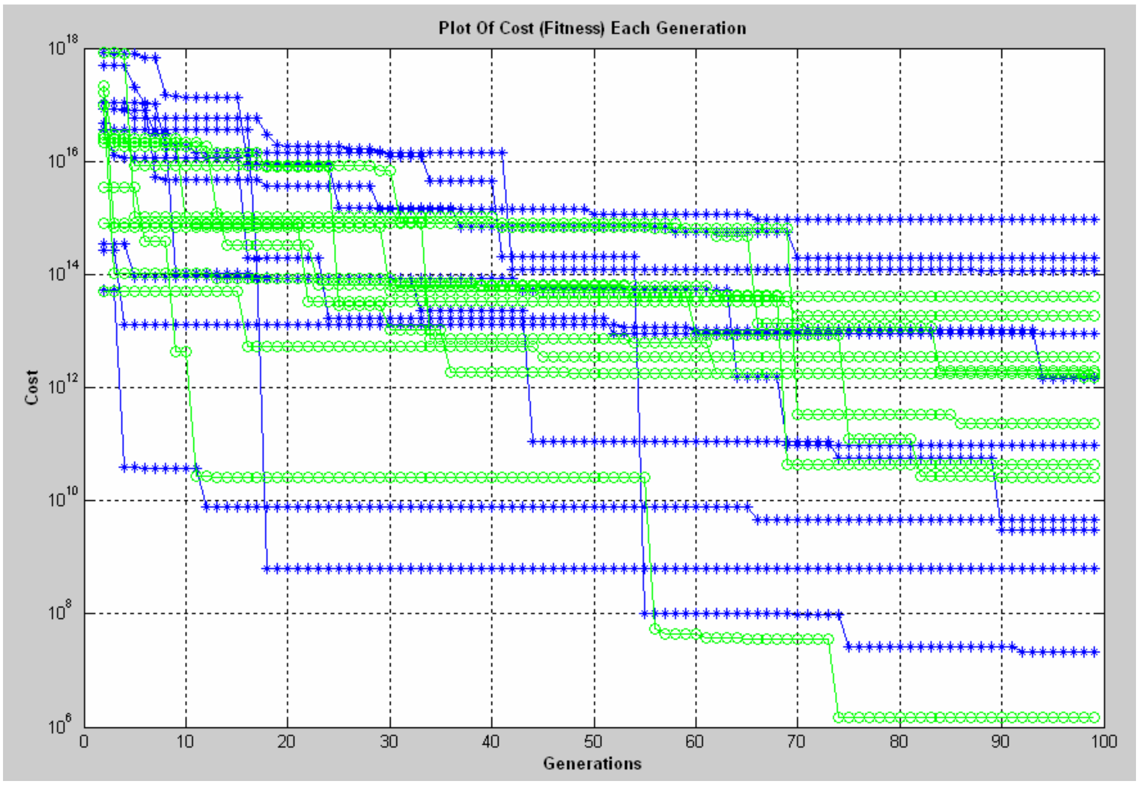


Figure 6-1 Semi-Log Cost Plot Varying Crossover Points

Table 6-2 Legend for Figure 6-1

Parameter	**Simulation A**	**Simulation B**
Plot line	Blue Stars	Green Circles
Simulation Runs	10	10
Generations	100	100
Crossover Points	**2**	**1**
Distribution	Normal	Normal
Mutation Rate	0.001	0.001
%Keep	70%	70%
Crossover Rate	0.5	0.5
Same Minimum	20	20

6.2.3.1.2 Base Performance Testing – Test A2

The Genetic Algorithm performed reasonably well after only 100 generations. Figure 6-2 shows that there is no significant advantage to optimizing the test control system by varying the distribution method using either a normal or uniform random distribution.

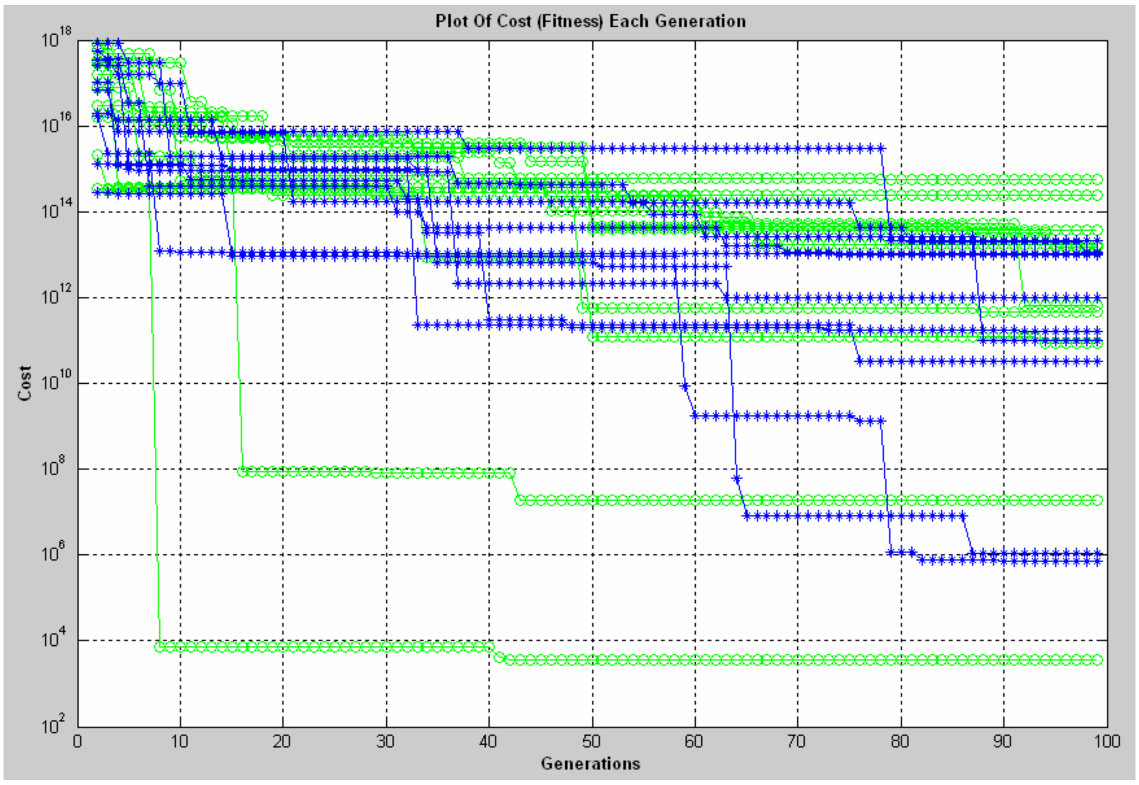


Figure 6-2 Semi-Log Cost Plot Varying Distribution

Table 6-3 Legend for Figure 6-2

Parameter	**Simulation A**	**Simulation B**
Plot line	Blue Stars	Green Circles
Simulation Runs	10	10
Generations	100	100
Crossover Points	1	1
Distribution	**Normal**	**Uniform**
Mutation Rate	0.001	0.001
%Keep	70%	70%
Crossover Rate	0.5	0.5
Same Minimum	20	20

6.2.3.1.3 Base Performance Testing – Test A3

The Genetic Algorithm performed reasonably well after only 100 generations. Figure 6-3 shows that varying `NKEEP` varies the result significantly. Clearly a 90% `NKEEP` does not provide enough diversification in the population in order to allow successful evolution. Evolution still occurs but at a retarded rate as most of the population remains unchanged.

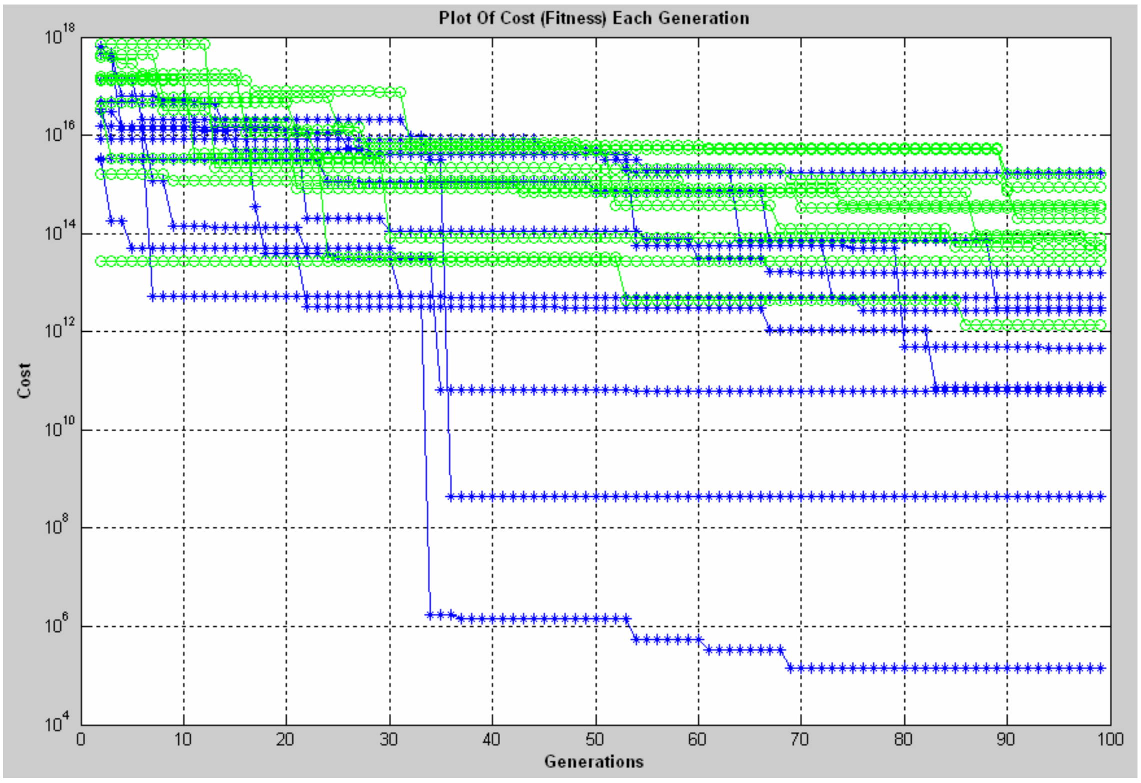


Figure 6-3 Semi-Log Cost Plot Varying NKeep

Table 6-4 Legend for Figure 6-3

Parameter	Simulation A	Simulation B
Plot line	Blue Stars	Green Circles
Simulation Runs	10	10
Generations	100	100
Crossover Points	1	1
Distribution	Uniform	Uniform

Mutation Rate	0.001	0.001
%Keep	**70%**	**90%**
Crossover Rate	0.5	0.5
Same Minimum	20	20

6.2.3.1.4 Base Performance Testing – Test A4

The Genetic Algorithm performed reasonably well after only 1000 generations with a least cost value of 164, as compared to the optimum value of 98.8382. Figure 6-4 shows that there is no significant advantage to optimizing the test control system by varying Crossover Rate.

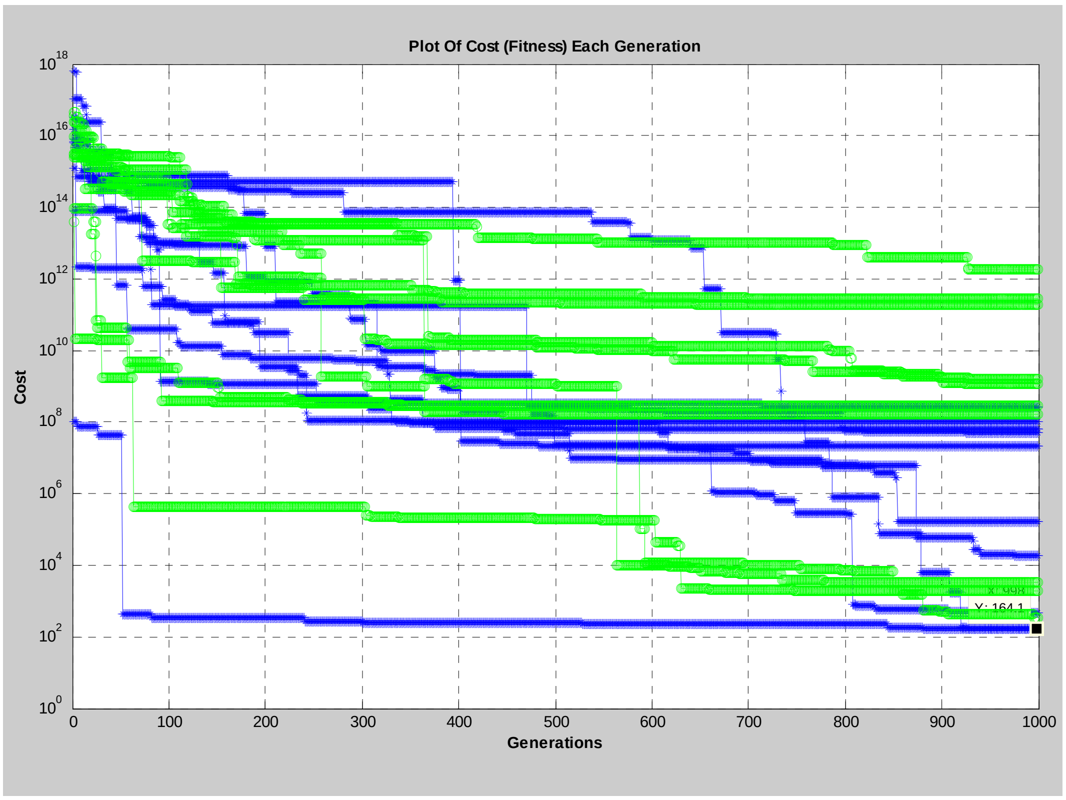


Figure 6-4 Semi-Log Cost Plot Varying Crossover Rate

Table 6-5 Legend for Figure 6-4

Parameter	**Simulation A**	**Simulation B**
Plot line	Blue Stars	Green Circles

Simulation Runs	10	10
Generations	100	100
Crossover Points	1	1
Distribution	Uniform	Uniform
Mutation Rate	0.001	0.001
%Keep	70%	70%
Crossover Rate	**0.75**	**0.5**
Same Minimum	20	20

6.2.3.1.5 Base Performance Testing – Test A5

The Genetic Algorithm performed reasonably well after only 1,000 generations. Figure 6-5 shows that varying the Mutation Rate varies the result significantly. Clearly a 10% Mutation Rate provided more diversification in the population which for this test control system allowed it to evolve more quickly than at a 0.1% rate. Evolution still occurs using the lower Mutation Rate but at a retarded rate because most of the population remains unchanged. This result is somewhat surprising as many background literature sources recommend a 0.1% Mutation Rate.

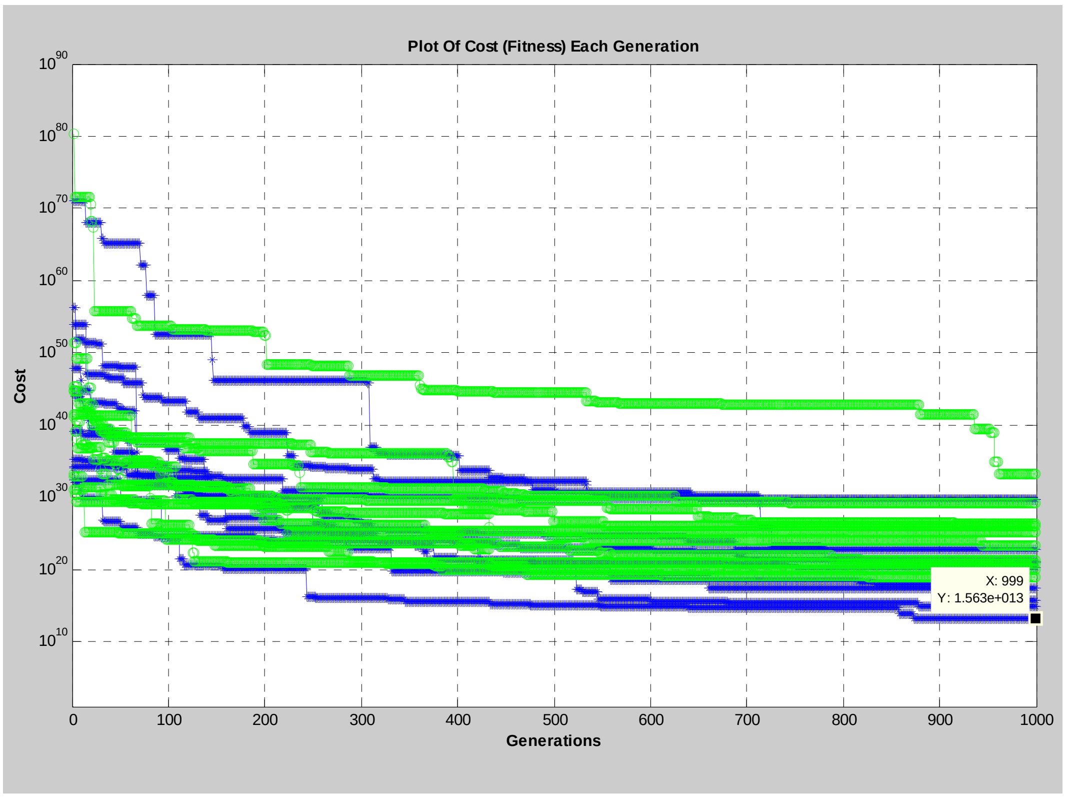


Figure 6-5 Semi-Log Cost Plot Varying Mutation Rate

Table 6-6 Legend for Figure 6-5

Parameter	**Simulation A**	**Simulation B**
Plot line	Blue Stars	Green Circles
Simulation Runs	10	10
Generations	1000	1000
Crossover Points	1	1
Distribution	Uniform	Uniform
Mutation Rate	**0.001**	**0.1**
%Keep	70%	70%
Crossover Rate	0.5	0.5
Same Minimum	20	20

6.2.3.2 Base Performance Testing – Phase B

Phase B testing aims to identify how well the Genetic Algorithm actually converges to the optimal result. To test this, ***Argo*** was simulated controlling both the first and second order systems. Because of the different rates of convergence, Phase B testing was run over 100,000 generations for three sample simulation runs using the first order test control system, then it was run over 10,000 generations for ten sample simulation runs using the second order test control system. The simulation results associated with the first order test control system are plotted in Figure 6-6 and listed in Table 6-7. The simulation results associated with the second order test control system are plotted in Figure 6-7 and listed in Table 6-8.

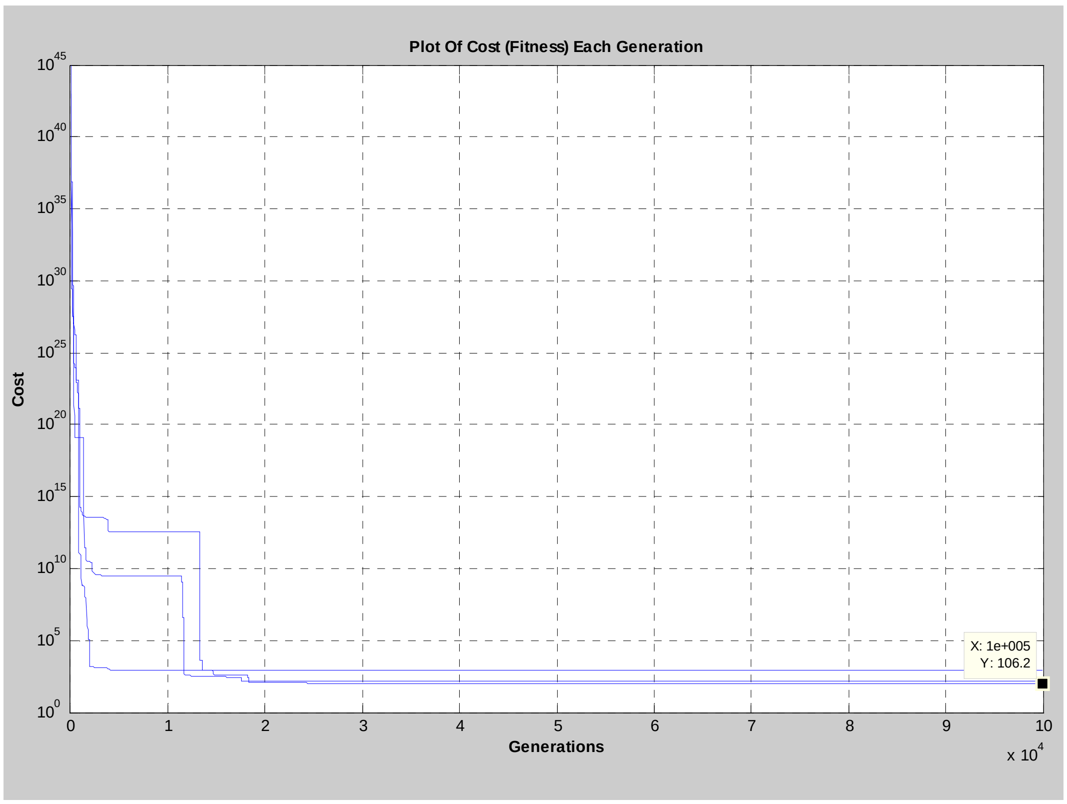


Figure 6-6 Semi-Log Cost Plot After 100,000 Generations

Table 6-7 Results for Phase C Base Performance Testing (First Order Test System)

	First Order Control System
Simulation	**Minimum Cost Value**
1	106.2222

2	793.4189
3	133.9407
Average	**106.2222**
Best	**344.5273**

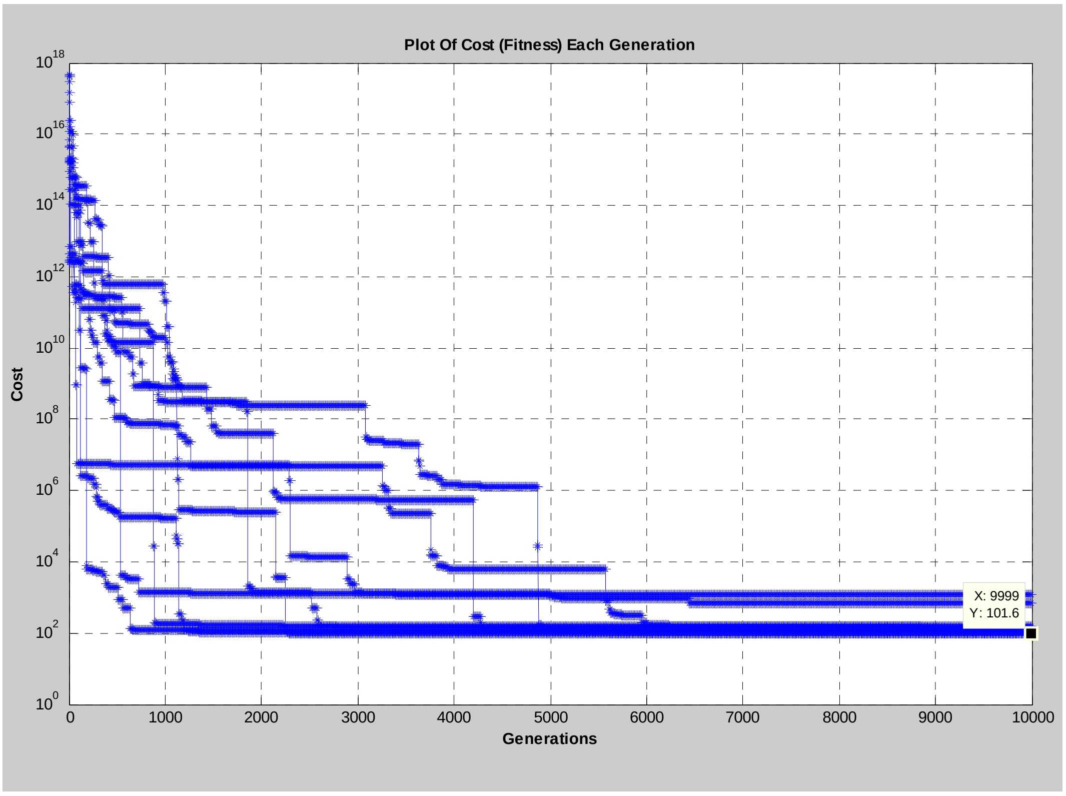


Figure 6-7 Semi-Log Cost Plot After 10000 Generations

Table 6-8 Results for Phase C Base Performance Testing (Second Order Test System)

	Second Order Control System
Simulation	**Minimum Cost Value**
1	150.1698
2	119.0186
3	113.4128

4	103.3038
5	686.0307
6	129.6848
7	100.4632
8	101.5698
9	1184.4477
10	151.0626
Average	**283.9164**
Best	**100.4632**

Although ***Argo*** failed to ever actually evolve the optimum set of PID parameters, on average, ***Argo*** did produce some reasonable solutions. It should be noted that these solutions may be suitable depending upon the actual performance requirements of the PID in terms of its practical application. Of the three samples for the first order test control system the best result was 106.2222. This is compared to the optimum result of 101.7664. Of the ten samples for the second order test control system the best result was 100.4632. This is compared to the optimum result of 99.4803.

6.2.3.3 Base Performance Testing – Phase C

Phase C testing aims to identify the number of generations required by ***Argo*** to achieve optimal results. The data collected in Phase B was deemed sufficient for analysis under Phase C. The data associated with the first order test control system is graphically represented in Figure 6-6 and listed in Table 6-7. The data associated with the second order test control system is graphically represented in Figure 6-7 and listed in Table 6-8. These plots showed that for the majority of samples ***Argo*** achieved its best results for the first order control system after approximately 25,000 generations, and approximately 5,000 generations for the second order test control system. Improvements to the population were minimal from these respective points onwards.

Statistically, improvement to ***Argo's*** results would continue if allowed to create further generations.

6.2.3.4 Base Performance Testing – Phase D

Phase D testing aimed to identify how quickly the Genetic Algorithm converges to its optimum results to identify suitability for practical applications. To test this, ***Argo's*** execution time was recorded over 10,000 generations and averaged to estimate a single generation's execution time. The results against the second order test control system are shown in Table 6-9. The experimental results were captured using an Intel^R Pentium^{R}M 1.86GHz processor on a Dell D610 laptop.

Table 6-9 ***Argo*** Execution Times

Generations	**Execution Time (seconds)**	**Calculation Method**	**Remarks**
10,000	1,740	Experimental	Large sample base
1	0.1740	Interpolation	Base computational cost
5,000	870	Extrapolation	Hypothesised execution time for achieving optimum results[2]

Thus, it can be hypothesised that for a reasonably optimum result at least 5,000 generations is required which would require a processing time of approximately 870 seconds (~13 minutes). Obviously this processing time would be vastly different for an embedded system.

6.3 Advanced *Argo* Operation

6.3.1 Conduct of Advanced Performance Testing

To analyse the advanced performance of the Genetic Algorithm the testing was conducted in three phases. The phases were:

[2] The number of generations required to achieve optimum results is based on the testing results described at Section 6.2.3.3.

a) **Phase E –** Comparing the Genetic Algorithm's performance using Roulette Wheel and Tournament selection methods (testing against the second order test control system only).

b) **Phase F –** Testing the rotary-wing control system using the Genetic Algorithm as the Optimiser of PID Controller Parameters using MATLAB®.

c) **Phase G –** Testing the Genetic Algorithm's performance controlling the first order test control system with a delayed input.

6.3.1.1 Advanced Performance Testing – Phase E

Phase E testing aimed to analyse the performance of the Genetic Algorithm using alternative selection methods. Specifically, Phase E testing aims to compare the performance of the Genetic Algorithm using both Tournament Selection and Roulette Wheel Selection. To test this, ***Argo's*** performance was compared using both selection methods over 10 simulation runs of 1,000 generations. Results against the second order test control system are shown in Figure 6-8. The comparison clearly shows that Tournament Selection out-performs Roulette Selection when controlling the second order test control system. Further, noting that the optimal cost value for the second order system is 98.8382, the Tournament Selection method achieved a best cost value of 128.5, compared with Roulette Selection method which achieved a best cost value in excess of 10^{11} both after 1,000 generations.

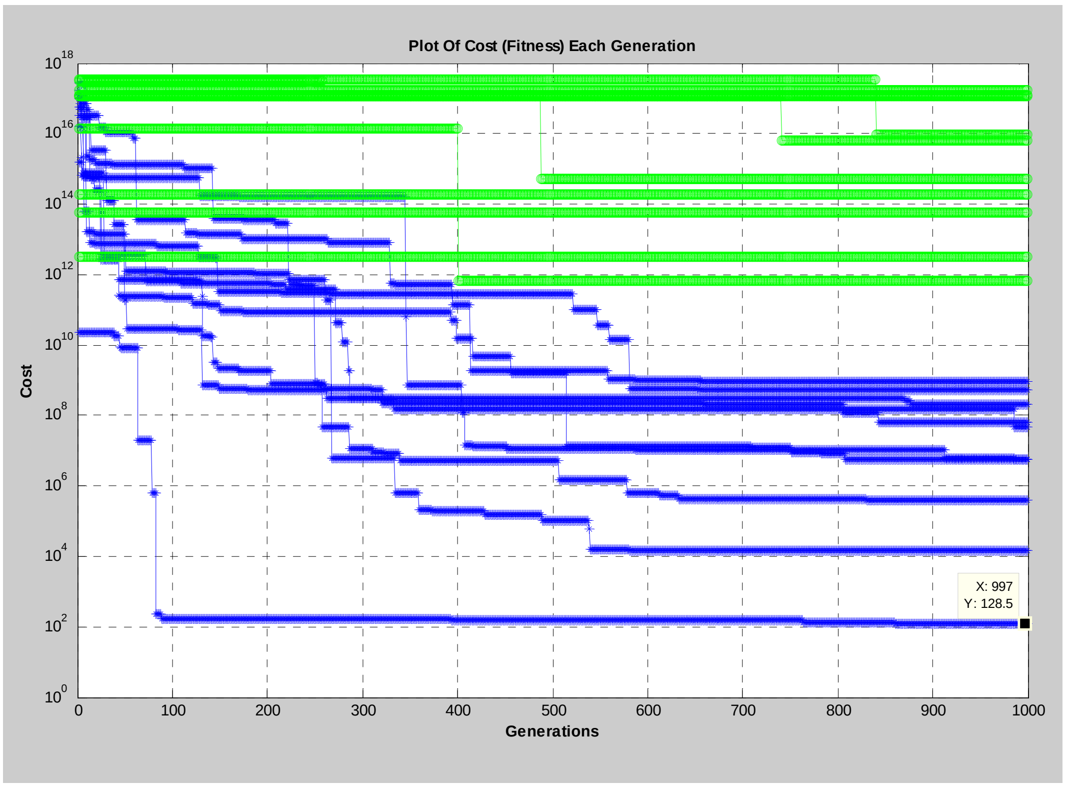


Figure 6-8 Semi-log Cost Plot Varying Selection Method

Table 6-10 Legend for Figure 6-8

Parameter	**Simulation A**	**Simulation B**
Plot line	Blue Stars	Green Plus Signs
Simulation Runs	10	10
Generations	1000	1000
Selection Method	**Roulette Wheel Selection**	**Tournament Selection**
Crossover Points	1	1
Distribution	Uniform	Uniform
Mutation Rate	0.1	0.1
%Keep	70%	70%
Crossover Rate	0.5	0.5

Same Minimum	20	20

6.3.1.2 Advanced Performance Testing – Phase F

Phase F testing was not conducted as the rotary-wing model was not integrated and simulated. This outcome was the result of two main factors: firstly, the processing time was considered impractically long; and secondly, the rotary-wing model was not available in MATLAB®, but only in SIMULINK® (conversion between the two formats was beyond the developer's ability at this point in the project).

6.3.1.3 Advanced Performance Testing – Phase G

Phase G testing aims to analyse the performance of the Genetic Algorithm controlling the first order test control system with delayed inputs. Specifically, Phase E testing aims to compare the performance of the Genetic Algorithm using five mean delayed input values of 1,3, 6, 10 and 30 samples (noting 6 represents the optimal delay) each with a standard deviation of 1. To test this, ***Argo's*** performance was compared over 3 simulation runs of 1,000 generations for each mean delayed input value.

However, before attempting to analyse the results from the simulation runs it is of value to examine how the system would be expected to behave in general. To provide this understanding two graphs (refer Figure 6-9 and Figure 6-10) have been provided to show the Output and Error response to the first order system using a short, optimum and long delay to the input. The short delay was based on 2 samples, optimum delay based on 6 samples and the long delay based on 10 samples.

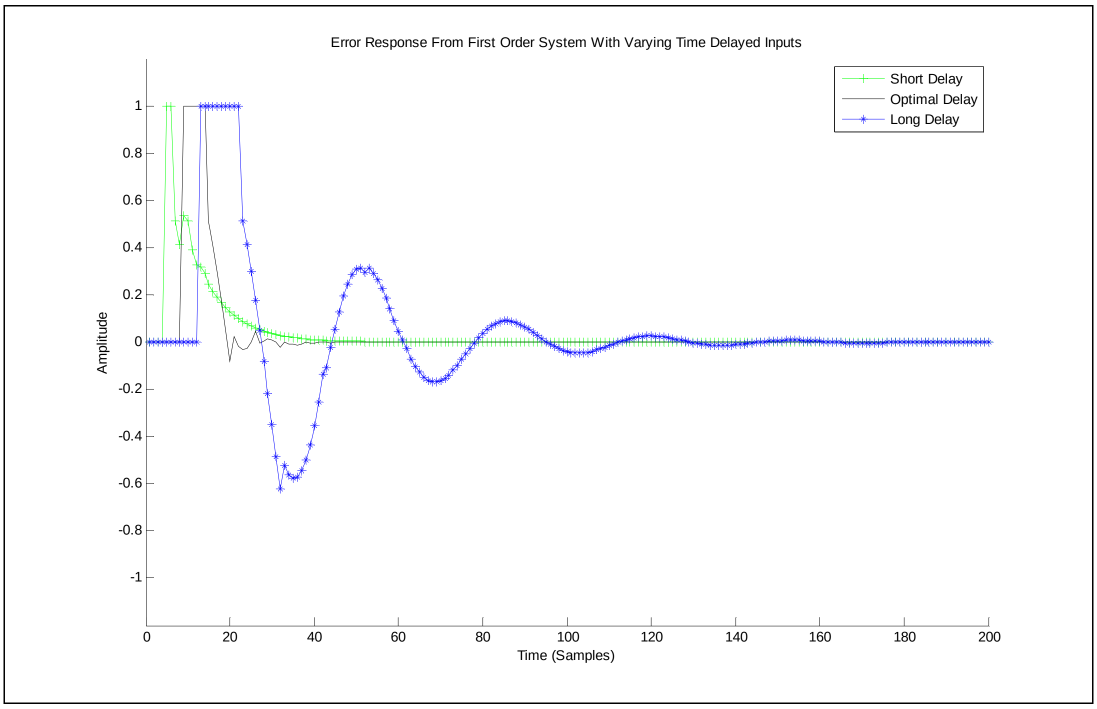


Figure 6-9 Error Response From First Order System With Varying Time Delayed Inputs

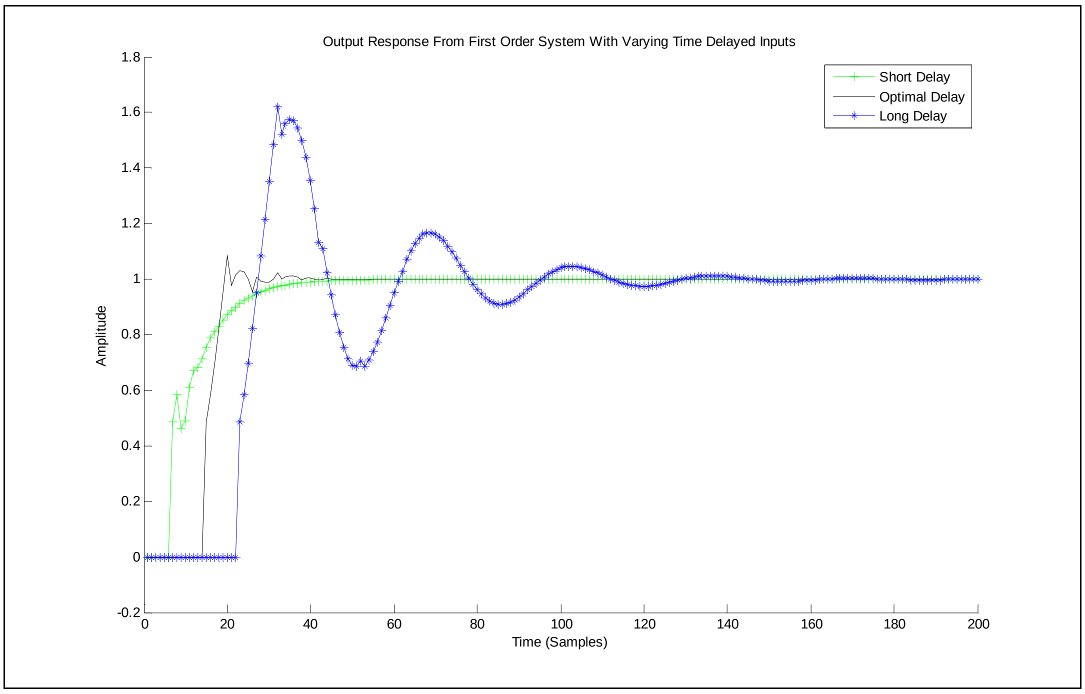


Figure 6-10 Output Response From First Order System With Varying Time Delayed Inputs

As can be seen from these graphs, the short delay produces an overdamped response in both cases. Likewise, the long delay produces an underdamped response. Both the short and optimum delays achieve steady state about the same time, whereas the long delay takes many more samples to achieve a stead state response.

It is important to then match this behaviour with the use of the ITAE Performance Criterion. That is, the ITAE function basically sums the area under the error curve (proportional to time) in order to provide a value representing the error. Hence, the long delay would be expected to display the worst ITAE values as the area under the curve for an underdamped response is the greatest.

Now that the system's expected behaviour has been considered, the actual test results may be analysed. The test results comparing the performance of each of the five delayed input scenarios using the first order test control system are shown in Figure 6-11. Unfortunately, the comparison graph shows the opposite of the expected response whereby the short delays return the greatest error and the longest delays return the least error. This is likely to be an artificial response due to the implementation of the `costfirstrandomdelay` function; most likely specific way in which the steepest descent minimisation routine handles the delayed samples in terms of the overall sample array. Regardless, the important behaviour to identify is that systems with delayed inputs will vary the error significantly (and hence cost) associated with the control system.

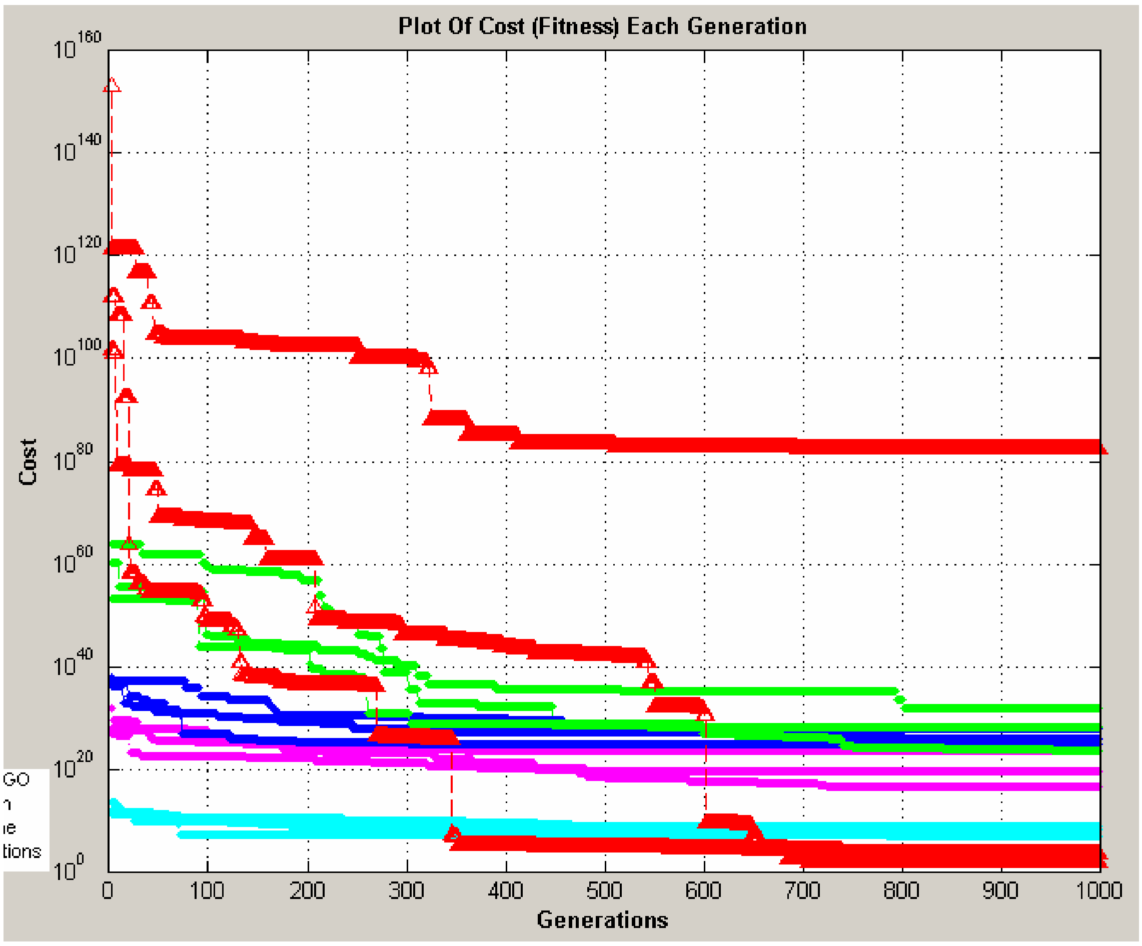


Figure 6-11 Semi-Log Cost Plot Varying Delayed Input Mean

Table 6-11 Legend for Figure 6-11

Parameter	**Delayed Input A**	**Delayed Input B**	**Delayed Input C**	**Delayed Input D**	**Delayed Input E**
Plot line	Red Triangle Point Line	Green Dashed Line	Blue Solid Line	Magenta Dotted Line	Cyan Dash-Dotted Line
Simulation Runs	3	3	3	3	3
Generations	1000	1000	1000	1000	1000

Delayed Input – Mean	**1**	**3**	**6**	**10**	**30**
Delayed Input – Standard Deviation	1	1	1	1	1
Selection Method	Tournament Selection	Tournament Selection	Tournament Selection	Tournament Selection	Tournament Selection
Crossover Points	1	1	1	1	1
Distribution	Uniform	Uniform	Uniform	Uniform	Uniform
Mutation Rate	0.1	0.1	0.1	0.1	0.1
%Keep	70%	70%	70%	70%	70%
Crossover Rate	0.5	0.5	0.5	0.5	0.5
Same Minimum	20	20	20	20	20

6.3.2 Other Test Observations

At the conclusion of the test program itself, two other general observations regarding the results have been made. The first regarding the operation of different plant systems, and secondly the operation of the ITAE Performance Criterion within the cost function.

As would be expected, different plant systems return significantly different cost plots. The second order control system was able to converge to a reasonably accurate result within 5,000 – 10,000 generations; whereas the first order control system required in the order of 100,000 generations to reach similar levels.

Secondly, the ITAE Performance Criterion used as the method for calculating the cost value also has an impact on the cost plot. That is, the ITAE value is basically a numerical representation of the area under the error curve (proportional to time). By its very nature, the error is also weighted proportional to the number of integrations used (that is, if a large number of iterations are used, the error at the beginning is weighted less than the error at the end). Thus, poor candidate solutions may have very high cost values; whereas good candidate solutions tend not to vary significantly. This problem is further exacerbated for the first order control system with delayed inputs. The result of a short input delay is an extremely large cost value (values in the order of 10^{140} were achieved for inputs delayed 5 samples from the optimum delay). This result suggests that although the ITAE Criterion may return large cost values for poor candidate solutions, its use may be quite appropriate for dealing with delayed inputs as it places more weight upon the steady state error (that is, the error associated with the latter iterations).

6.4 Summary

In summary, this chapter has presented and analysed the test results of the Genetic Algorithm program itself – ***Argo***. To do this the chapter presented the results against both the basic and advance specifications of the project.

6.4.1 Base Performance Testing Results

In brief, ***Argo*** performed reasonably well against the basic specifications, noting that it took significant processing time to achieve a satisfactory outcome. Typically ***Argo*** took in excess of 25 minutes to process 10,000 generations to achieve a result only one order of magnitude greater than the optimum value. Reasons for this significant processing cost are discussed further in the final chapter.

In terms of the parameter settings themselves, the test results show that in general the Genetic Algorithm itself is not sensitive to minor changes in parameter values.

6.4.2 Advanced Performance Testing Results

The test results also suggested that Tournament Selection was the superior selection method (over Roulette Wheel) for use with both test control systems.

Unfortunately, testing against the final advanced specification was not undertaken as a MATLAB® rotary-wing control model was not able to be sourced (only a SIMULINK® model was available).

Chapter 7

Conclusions

7.1 Dissertation Summary

In summary, the aim of this project was to design and code – using MATLAB® – an optimised PID controller using a Genetic Algorithm to perform the optimisation routine. The aim was then further broken down to establish four primary and two secondary objectives for the project:

Primary Objectives (Base Functionality)

Step 1. Research the background information relating to Genetic Algorithms.

Step 2. Design a Genetic Algorithm for implementation using a 3rd generation program language, specifically MATLAB®, within set specifications (refer Appendix A Project Specification).

Step 3. Code the designed Genetic Algorithm using MATLAB®.

Step 4. Test the Genetic Algorithm against specifications.

Secondary Objectives (Advanced Functionality)

Step 5. Increase the functionality of the Genetic Algorithm through the addition of a user option to configure for Roulette Wheel based selection.

Step 6. Model the Genetic Algorithm for use controlling a rotary-wing control system using MATLAB® (MATLAB® rotary-wing model to be provided by the Project Supervisor).

In summarising the projects performance, this chapter is structured into three main sections: Basic ***Argo*** Operation; Advanced ***Argo*** Operation; and Further Work. In doing so, the paper critiques the project's successes as well as shortcomings.

7.2 Basic *Argo* Operation

7.2.1 Accuracy

The results achieved by using ***Argo*** to control the test control systems indicate that convergence to the known optimal solutions was reasonably successful. However, under no test conditions did ***Argo*** actually converge to the optimal solution. Further, the absolute error from the known optimal solution varied greatly with each run. Whilst the results overall were not as positive as first hoped, the results must be viewed in perspective of total operation. That is, ***Argo*** has an extremely large solution space whereby each chromosome could be assigned any value in the gene's range (refer EQN 7–1):

$$2^{30} \Rightarrow -1{,}073{,}741{,}823 < q_n < +1{,}073{,}741{,}823$$

EQN 7–1

When the range of each gene and then chromosome is considered along with the population size of 50, it is perhaps not surprising that convergence to the known optimum results were not achieved within 10,000 generations.

As the test results indicated, different plant systems also resulted in vastly different cost plots. The second order control system was able to converge to a reasonably accurate result within 5,000 – 10,000 generations; whereas the first order control system required in the order of 100,000 generations to reach similar levels.

7.2.2 Processing Time

The results presented in Chapter 6 were achieved with significantly high processing times. This penalty is undoubtedly due to the high computational cost of the Genetic Algorithm itself. Typically, 1 simulation of 10,000 generations within a population size of 50 took approximately 26 minutes to process (refer Table 6-10).

Although this significantly high processing time could be directly caused because of the specific nature and design of ***Argo*** itself, the result does suggest that Genetic Algorithms are not suitable to all practical applications. That is, it would still appear to be more practical to use classical analytical optimisation techniques for problems that can be easily solved as such; whereas Genetic Algorithm optimisation, regardless of time penalty, may be more suitable for complex problems not able to be solved using classical methods. Interestingly though, the long processing times associated with Genetic Algorithms are mimicked within Nature – philosophically Genetic Algorithms simulate the process of evolution, not revolution!

It is also noted that the application of standard coding optimisation techniques (such as the use of registers vice variables) may result in improved processing times for ***Argo***. Undoubtedly, the use of MATLAB®'s inherent vectorisation ability could also be used more effectively within ***Argo*** – especially considering the number of iterative programming loops used and the number of array operations. It is also possible that the use of an alternate programming language could see improvements in processing time. Additionally, the modification of the Genetic Algorithm to employ a RGO approach (refer Section 1.4.1) may also improve processing time by reducing the solution space. Finally, the deployment of the Genetic Algorithm to an embedded system will also likely improve the computational efficiency significantly.

7.2.3 Basic Parameters

The results as presented in Chapter 6 suggest that the Genetic Algorithm is not very sensitive to **minor** changes in the basic parameters such as `NUMBER_OF_CROSSOVER_POINTS, DISTRIBUTION, NKEEP, CROSSOVER_RATE or MUTATION_RATE`. This indicates that no single parameter has optimum or ideal values for operation as part of a Genetic Algorithm. There would however appear to be a range outside of which that each parameter would cause undesirable disruption within the Genetic Algorithm's operation. If the Genetic Algorithm is too disruptive it may converge at a slower rate. For example, clearly a mutation rate around 1% is ideal: less than 1% results in retarded convergence rates; likewise greater than 1% cases too much disruption and again results in retarded convergence rates.

7.3 Advanced *Argo* Operation

7.3.1 Roulette Wheel Selection

The test results using the second order test control system clearly indicated that the Tournament Selection method produced superior results to the Roulette Wheel Selection method. This result was surprising as Roulette Wheel based selection would appear to be the most commonly used method across the background literature reviewed. Conceptually though, Tournament Selection does more closely mimic nature in operation. Also, Tournament Selection does not require ranked sorting of each generation's population, thus avoiding a significant computational time penalty. Hence this paper makes the recommendation to use a Tournament Selection method for practical Genetic Algorithm applications.

7.3.2 Rotary-Wing Control Model Simulation

Unfortunately ***Argo*** was not able to be simulated controlling the rotary-wing control model. This outcome was the result of two main factors: firstly, the processing time was considered impractically long; and secondly, the rotary-wing control model was not available in MATLAB®, but only in SIMULINK® (conversion between the two formats was beyond the developer's ability at this point in the project). Whilst the long processing times make adaptive tuning impractical, theoretically it is still possible (dependent upon possession of a MATLAB® based model) to integrate and simulate the rotary-wing control model in a fixed tuning configuration (refer Section 3.3). Interestingly, the need to employ adaptive tuning would depend upon the practical application. It is likely that for most practical applications a well optimised fixed-tuned configuration digital control system would suffice – the very nature of the feedback design and PID controller should force an acceptable output signal in most situations barring a rapidly changing operating environment.

7.4 Further Work

Following completion of the project and presentation of results, this paper makes five suggestions for areas that could be undertaken as further work to the project. Firstly, the opportunity clearly exists for the optimisation of the current MATLAB® code. As suggested in Section 7.2.2 this could be further extended to alternate programming languages and development of an embedded GA optimisation system. Secondly, the opportunity still remains for the integration and simulation of rotary-wing model in SIMULINK®. Thirdly, in terms of simplification of operation, an improved GUI could prove a simple and effective extension to the application (refer Section for current interface design). The fourth opportunity would be to attempt to modify the Genetic Algorithm to employ a RGO approach in an attempt to improve processing time. Finally, from a purely research perspective in the field of Genetic Algorithms, the challenge of developing a set of *Rules of Thumb* for the configuring of Genetic Algorithm parameters could be undertaken. This would greatly assist developers of practical Genetic Algorithm applications.

Finally, whilst this project has successfully designed and implemented a Genetic Algorithm to optimise a PID Controller, it can be seen that there remains many improvements and challenges before the technology could be practically deployed to industry. However, as the field itself matures, it is expected that this concept could yet see real success.

List of References

AAAI 2000-2005, *Genetic Algorithm: A Subtopic of Machine Learning*, online, <http://www.aaai.org/AITopics/html/genalg.hml>

AS/NZS 4360:2004 *Risk Management*, Australian and New Zealand Standard.

Buckland, M (2005a), *AI-Junkie: Genetic Algorithms in Plain English*, online, http://wwww.ai-junkie.com/ga/intro/gat1.html

Buckland, M (2005b), *Basic Genetic Algorithm – ga_tutorial.cpp*, <http://www.ai-junkie.com/ga/intro/gat3.html>

Cantu-Paz, E (2001), *Genetic Algorithms and Evolutionary Computation: Efficient and Accurate Parallel Genetic Algorithms*, Second Printing, Kluwer Academic Publishers, USA, Massachusetts.

Chambers, L (ed) 1995, *Practical Handbook of Genetic Algorithm Applications*, vol. 1, CRC Press, Florida.

DeJong, K (1975), *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, PhD Dissertation. Department of Computer and Communication Sciences, University of Michigan, Ann Arbor.

Dorf, RC (1992), *Modern Control Systems*, 6th edn, Addison-Wesley, Sydney.

ELE3105 Computer Controlled Systems: Study Guide (2007), University of Southern Queensland, Toowoomba.

Garrido, S, Moreno, L & Salichs, MA (n.d.), *Predictive Control With Restricted Genetic Optimization*, Research Paper, Universidad Carlos II de Madrid, Spain.

Haupt, RL & Haupt, SE 2004, *Practical Genetic Algorithms*, 2nd edn, Wiley-Interscience, Canada.

Holland, J 1975, *Adaptation In Natural and Artificial Systems*. University of Michigan Press.

Hunter, J 2002, *Encyclopedia Mythica™: Argo*, online, http://www.pantheon.org/articles/a/argo.html

Land, B 2007, *BIONB441 Biological Instrumentation: Matlab Graphical User Interface Design.* Cornell University, online, http://courses.cit.cornell.edu/bionb442/GUIdesign/GUIdemo.m

Leis, J 2003, *ELE3105 Computer Controlled Systems: Course Notes*, University of Southern Queensland, Toowoomba.

Mackenzie, M (2004), *ELE 3105 Computer Controlled Systems: Assignment 2*, University of Southern Queensland, Toowoomba.

Merriam-Webster's Medical Desk Dictionary, 2002, Revised Edition, Merriam-Webster, Inc., Dictionary.com website, viewed 27 Nov 2006, http://dictionary.reference.com/search?db=mwmed&q=centromere

The American Heritage® Dictionary of the English Language, 2004, Fourth Edition, Dictionary.com website, viewed 27 Nov 2006, http://dictionary.reference.com/browse/Meiosis

The American Heritage® Stedman's Medical Dictionary, 2006, Fourth Edition, Dictionary.com website, viewed 27 Nov 2006, http://dictionary.reference.com/browse/Meiosis

Whitley, W (n.d.), *A Genetic Algorithm Tutorial*, Computer Science Department, Colorado State University, Colorado.

Bibliography

The GP Tutorial, 1996-2003, Members of the Castles of the World Network, viewed 30 Jan 05, <http://geneticprogramming.com/Tutorial/>

The Institution of Engineers, Australia (2000), *Code of Ethics*, online, <http://www.engineersaustralia.org.au/index.cfm>

Towards Sustainable Engineering Practice: Engineering Frameworks for Sustainability, Institution of Engineers, Australia, Canberra, 1997.

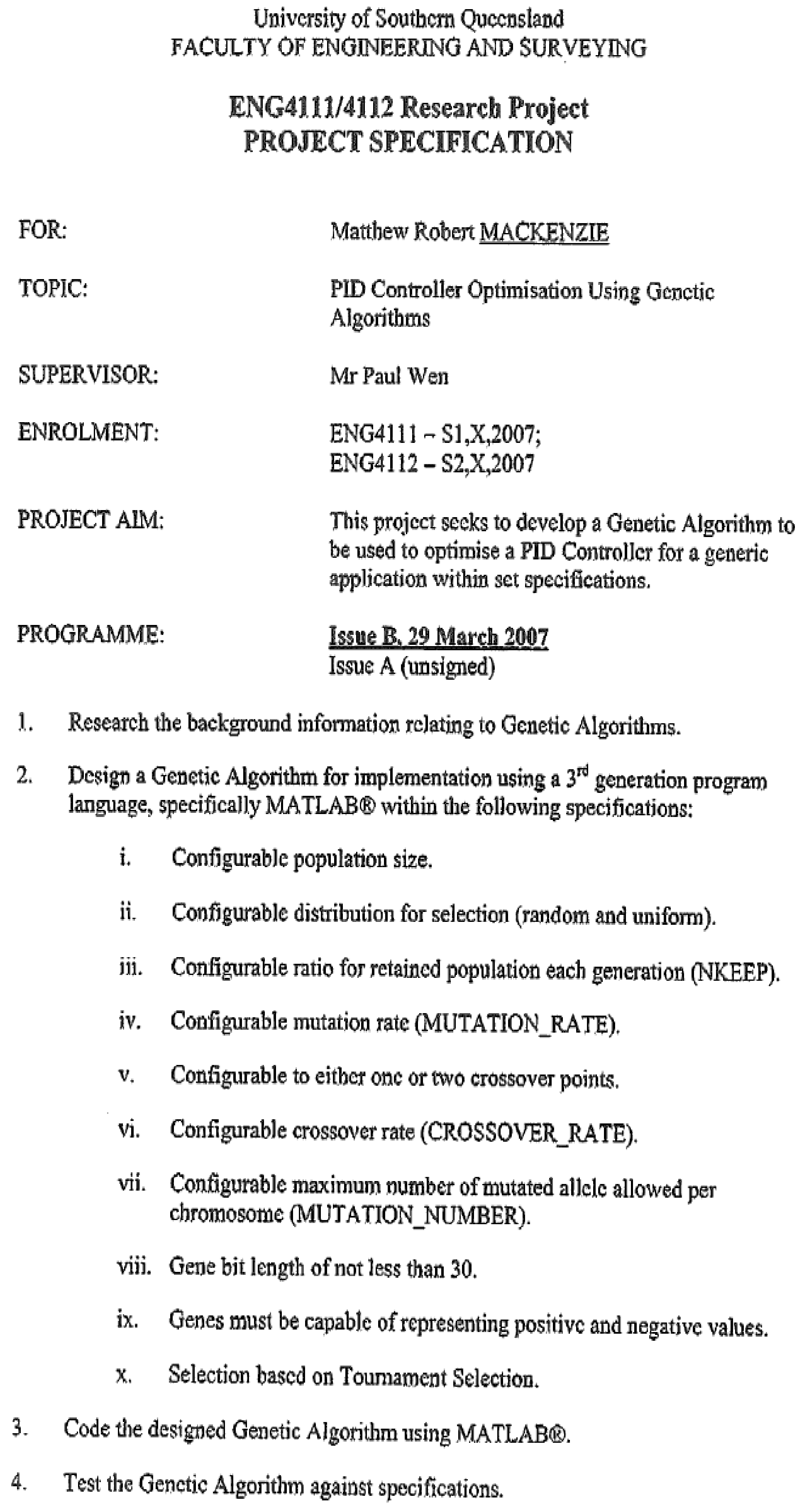
Wall, M (n.d.), *Overview of Genetic Algorithms*, viewed 30 Jan 05, <http://lancet.mit.edu/~mbwall/presentations/IntroToGAs/P001.html>

Wright, A (n.d.), *Genetic Algorithms for Real Parameter Optimisation*, Department of Computer Science, University of Montana.

Appendix A

Project Specification

A signed copy of the Project Specification is provided on the following page.

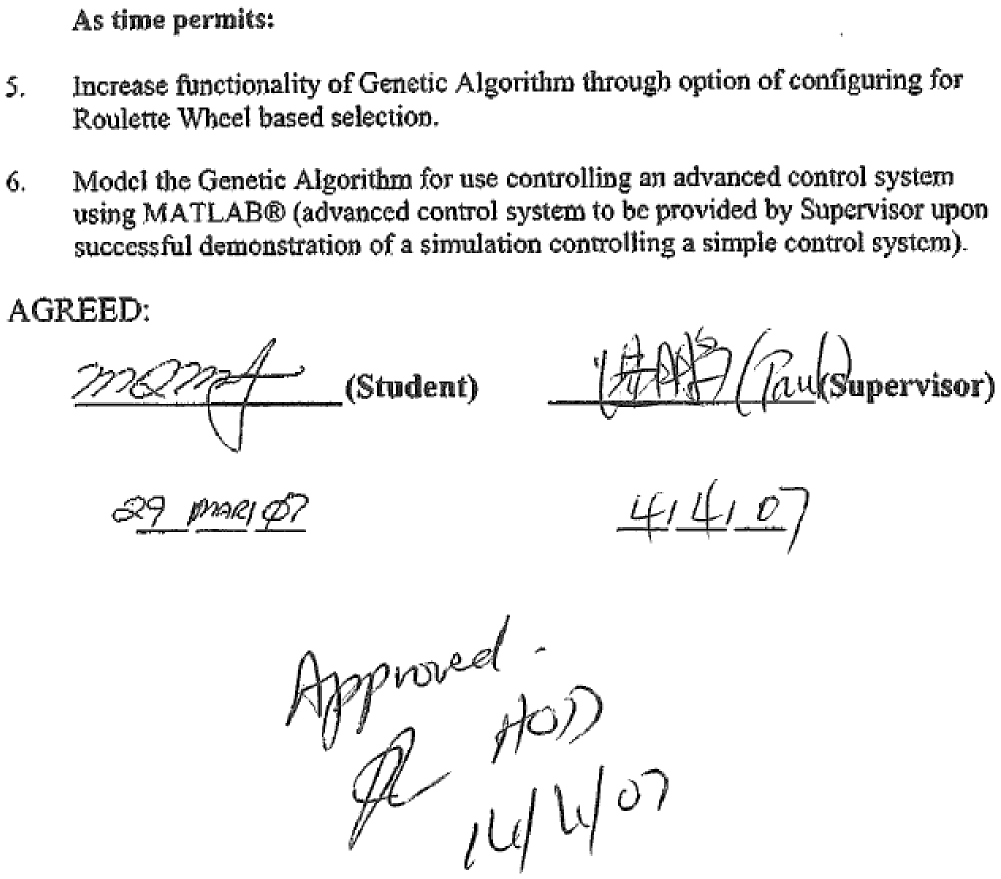


University of Southern Queensland
FACULTY OF ENGINEERING AND SURVEYING

ENG4111/4112 Research Project
PROJECT SPECIFICATION

FOR:	Matthew Robert <u>MACKENZIE</u>
TOPIC:	PID Controller Optimisation Using Genetic Algorithms
SUPERVISOR:	Mr Paul Wen
ENROLMENT:	ENG4111 – S1,X,2007; ENG4112 – S2,X,2007
PROJECT AIM:	This project seeks to develop a Genetic Algorithm to be used to optimise a PID Controller for a generic application within set specifications.
PROGRAMME:	**<u>Issue B, 29 March 2007</u>** Issue A (unsigned)

1. Research the background information relating to Genetic Algorithms.
2. Design a Genetic Algorithm for implementation using a 3rd generation program language, specifically MATLAB® within the following specifications:
    - i. Configurable population size.
    - ii. Configurable distribution for selection (random and uniform).
    - iii. Configurable ratio for retained population each generation (NKEEP).
    - iv. Configurable mutation rate (MUTATION_RATE).
    - v. Configurable to either one or two crossover points.
    - vi. Configurable crossover rate (CROSSOVER_RATE).
    - vii. Configurable maximum number of mutated allele allowed per chromosome (MUTATION_NUMBER).
    - viii. Gene bit length of not less than 30.
    - ix. Genes must be capable of representing positive and negative values.
    - x. Selection based on Tournament Selection.
3. Code the designed Genetic Algorithm using MATLAB®.
4. Test the Genetic Algorithm against specifications.



As time permits:

5. Increase functionality of Genetic Algorithm through option of configuring for Roulette Wheel based selection.

6. Model the Genetic Algorithm for use controlling an advanced control system using MATLAB® (advanced control system to be provided by Supervisor upon successful demonstration of a simulation controlling a simple control system).

AGREED:

[illegible] **(Student)** [illegible] (Paul) **(Supervisor)**

29 MAR 07 4/4/07

Approved -
HOD
[illegible]
14/4/07

Appendix B

MATLAB® Source Code – *Argo*

The MATLAB code for the following functions is found in the corresponding Annexes:

a) Annex A to
Appendix B
Argo – Main Routine

b) Annex B to
Appendix B
Argo – Genetic Algorithm

c) Annex C to
Appendix B
Argo – InitialisePopulation

d) Annex D to
Appendix B
Argo – TournamentSelection

e) Annex E to
Appendix B
Argo – ParseBits

f) Annex F to
Appendix B
Argo – RouletteWheel

g) Annex G to
Appendix B
Argo – Mutate

h) Annex H to
Appendix B
Argo – Crossover

i) Annex I to
Appendix B
Argo – CalcFitness

j) Annex J to
Appendix B
Argo – CostFirst

k) Annex K to
Appendix B
Argo – CostFirstRandomDelay

l) Annex L to
Appendix B
Argo – CostSecond

m) Annex M to
Appendix B
Argo – ARGOGUI

Annex A to Appendix B *Argo* – Main Routine

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE4111/2
%   Student No:     Q9323707
%
%   ARGOR.m interfaces with the Genetic Algorithm in order to
%   optimise the parameters q0,q1 and q2 for a PID controller.
%
%   INPUT   user can set parameters for:
%           displaying comments
%           selection method, either 'Tournament' or
%               'Roulette Wheel'
%           simulation runs
%           number of crossover points, either 1 or 2
%           distribution type, either 'uniform' or 'normal'
%           mutation rate
%           mutations per chromosome
%           maximum generations allowed
%           percentage of population to keep each generation
%           crossover rate
%           average of random delay input
%           standard deviation of random delay input
%           maximum same minimum cost value allowed in
%               consecutive generations
%
%   OUTPUT  semi log plot of the minimum costs within each
%           generation
%
%   Updated 18 Sep 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

% SET CONSTANTS
TRUE=1;
FALSE=0;

% SET INFO
INFO=TRUE; %TRUE = comments ON, FALSE = comments OFF

% DISPLAY INFORMATION
if (INFO == TRUE)
    fprintf('\n\n\n\n============ARGO===========\n');
    fprintf('Welcome to ARGO, a Genetic Algorithm for finding the
\n');
    fprintf('optimum values for parameters q0, q1 and q2 for a
PID\n');
    fprintf('controller.\n\n');
%    fprintf('STATUS ... UNDER TEST\n\n');
    fprintf('STATUS ... PROTOTYPE\n\n');
%    fprintf('STATUS ... OPERATIONAL\n\n');
    fprintf('Written by Matthew Mackenzie\n\n');
else
    fprintf('ARGOS\n');
end

% SET PARAMETERS
```

```
SELECTION='Tournament';
SIMULATION_RUNS=1;
NUMBER_OF_CROSSOVER_POINTS=1;
DISTRIBUTION='uniform';
MUTATION_RATE=0.1;
MUTATIONS_PER_CHROMOSOME=1;
MAX_GENERATIONS=500;
NKEEP=.7;
CROSSOVER_RATE=.5;
AVERAGE=3;
STDDEV=1;
SAME_MIN=20;
                    % var for convergence testing - #times can
continue
                    % with same min
COMMENTS=FALSE; %TRUE = comments ON, FALSE = comments OFF
if (INFO == TRUE)
    fprintf('SELECTION = %s\n',SELECTION);
    fprintf('SIMULATION_RUNS = %d\n',SIMULATION_RUNS);
    fprintf('NUMBER_OF_CROSSOVER_POINTS =
%d\n',NUMBER_OF_CROSSOVER_POINTS);
    fprintf('DISTRIBUTION = %s\n',DISTRIBUTION);
    fprintf('MUTATION_RATE = %f\n',MUTATION_RATE);
    fprintf('MUTATIONS_PER_CHROMOSOME =
%d\n',MUTATIONS_PER_CHROMOSOME);
    fprintf('MAX_GENERATIONS = %d\n',MAX_GENERATIONS);
    fprintf('NKEEP = %f\n',NKEEP);
    fprintf('CROSSOVER_RATE = %f\n',CROSSOVER_RATE);
    fprintf('SAME_MIN = %d\n',SAME_MIN);
    fprintf('Parameters set ...\n\n');
end

% INITIALISE VARIABLES
MATRIX=zeros(SIMULATION_RUNS,MAX_GENERATIONS-1);

for z = 1:SIMULATION_RUNS,

    fprintf('Current simulation run is = %d\n',z);
    % each simulation run must have the same random delay for the
control
    % system in order to compare results against a common baseline
    Random_Delay=round(AVERAGE+STDDEV.*randn);
    % CALL GENETIC ALGORITHM FUNCTION

[New_Best_Mag,gene1dec,gene2dec,gene3dec]=GeneticAlgorithmRandomDelays
(SELECTION,NUMBER_OF_CROSSOVER_POINTS,DISTRIBUTION,MUTATION_RATE,MUTAT
IONS_PER_CHROMOSOME,MAX_GENERATIONS,NKEEP,CROSSOVER_RATE,SAME_MIN,COMM
ENTS,Random_Delay);

    % OUTPUT SUMMARY
    fprintf('\n========OUTPUT SUMMARY======\n');
    fprintf('q0 %g\n q1 %g\n q2 %g \n\n',gene1dec,gene2dec,gene3dec);
    figure(1);
    semilogy(New_Best_Mag,'b*-');
    fprintf('\n========END OF SUMMARY=======\n\n\n\n\n');
    title('\bfPlot Of Cost (Fitness) Each Generation');
    grid on;
    hold on;
    xlabel('\bfGenerations');
    ylabel('\bfCost');

    %   matrix of minimum costs each generation and each simulation
    %   run kept for data collection purposes
    MATRIX(z,:)=New_Best_Mag;

end

% MATRIX
```

Annex B to Appendix B *Argo* – Genetic Algorithm Random Delay

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE4111/2
%   Student No:     Q9323707
%
%   GeneticAlgorithmRandomDelays.m performs Genetic Algorithm
%   selection to optimise the parameters q0,q1 and q2 for a
%   PID controller.
%
%   The C++® source code provided online by:
%
%     Buckland, M (2005b) Basic Genetic Algorithm – ga_tutorial.cpp,
%     <http://www.ai-junkie.com/ga/intro/gat3.html>
%
%   was used to provide a conceptual template for how the main
%   routine could be structured within any practical Genetic
%   Algorithm. This conceptual framework was used as a starting
%   point and subsequently extensively modified and built upon to
%   meet the specific requirements of Argo itself.
%
%   INPUT   SELECTION:  selection method, either 'Tournament'
%                       or 'Roulette Wheel'
%           NUMBER_OF_CROSSOVER_POINTS: the number of crossover
%                       points to be used during mating
%           DISTRIBUTION: the random distribution function, either
%                       'normal' or 'uniform'
%           MUTATION_RATE: the rate of mutation for allele
%           MUTATIONS_PER_CHROMOSOME: the maximum number of
%                       mutations allowed on a single chromosome
%                       following mating
%           MAX_GENERATIONS: the maximum generations allowed
%                       before termination of the program
%                       should convergence not occur
%           NKEEP:      the percentage of the population kept
%                       each generation
%           CROSSOVER_RATE: the liklihood of crossing over
%                       genetic codes between mating parents
%           SAME_MIN:   the maximum number of times no improvement
%                       between generations can occur before the
%                       genetic algorithm will terminate
%           COMMENTS:   boolean to denote if comments are on or off
%           random_delay: the random delay to input
%
%   Updated 18 Sep 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

function
[New_Best_Mag,gene1dec,gene2dec,gene3dec]=GeneticAlgorithmRandomDelays
(SELECTION,NUMBER_OF_CROSSOVER_POINTS,DISTRIBUTION,MUTATION_RATE,MUTAT
IONS_PER_CHROMOSOME,MAX_GENERATIONS,NKEEP,CROSSOVER_RATE,SAME_MIN,COMM
ENTS,random_delay)

% Set Constants
GENE_LENGTH=30;
CHROMO_LENGTH=93;
```

```
POP_SIZE=50;
FALSE=0;
TRUE=1;
if (COMMENTS == TRUE)
    fprintf('GENE_LENGTH = %d\n',GENE_LENGTH);
    fprintf('CHROMO_LENGTH = %d\n',CHROMO_LENGTH);
    fprintf('POP_SIZE = %d\n',POP_SIZE);
    fprintf('FALSE = %d\n',FALSE);
    fprintf('TRUE = %d\n',TRUE);
    fprintf('Constants set ...\n\n');
end

% Initialise Variables
Generations=2; % index into which generation current operation is on,
                   % the first generation is the initialised
generation
Min_array=zeros(1,MAX_GENERATIONS); % global statistic for convergence
testing
Mean_array=zeros(1,MAX_GENERATIONS); % global statistic for
convergence testing
Stop=FALSE; % set main loop to start for begining of first run
SmallerMin=TRUE;
SmallerMean=TRUE;
SameMin=0;
SameMean=0;
New_Best_Mag=0;
New_Best_Index=0;
New_Best_Chromo=0;
Old_Best_Mag=0;
Old_Best_Index=0;
Old_Best_Chromo=0;
Fitness_Matrix=zeros(POP_SIZE);
if (COMMENTS == TRUE)
    fprintf('Variables initialised ...\n\n');
end

% INITIALISE POPULATION
[population]=InitialisePopulation(POP_SIZE,GENE_LENGTH);
if (COMMENTS == TRUE)
    fprintf('Population initialised ...\n\n');
    fprintf('Commencing main routine, please wait ...\n\n');
end

% CALCULATE FIRST FITNESS VALUES
[Fitness_Array]=CalcFitnessRandomDelay(population,GENE_LENGTH,random_d
elay);
[Min_Fitness_Mag,Min_Fitness_Index,Mean_Fitness]=FitnessStats(Fitness_
Array);
Min_array(1)=Min_Fitness_Mag;   % store pop fitness statistics to
Mean_array(1)=Mean_Fitness;        % use for convergence testing
Fitness_Matrix=[Fitness_Matrix;Fitness_Array];

Old_Best_Index=Min_Fitness_Index;
Old_Best_Mag=Min_Fitness_Mag;
Old_Best_Chromo=population(Min_Fitness_Index,:);

while ((Generations < (MAX_GENERATIONS)) & (Stop == FALSE))

   % CREATE INTERMEDIATE POPULATION
   inter_pop_size=floor(POP_SIZE*NKEEP);
   IntermediatePopulation=population;
   for i=1:(POP_SIZE - inter_pop_size) % loop to replace non-surviving
chromosomes

      % SELECTION
      if (strcmpi(SELECTION,'Tournament')==1)

Selected_Chromo_Index_A=TournamentSelection(POP_SIZE,Fitness_Array);

Selected_Chromo_Index_B=TournamentSelection(POP_SIZE,Fitness_Array);
      elseif (strcmpi(SELECTION,'Roulette')==1)

Selected_Chromo_Index_A=RouletteWheel(POP_SIZE,Fitness_Array);

Selected_Chromo_Index_B=RouletteWheel(POP_SIZE,Fitness_Array);
      end
      chromo_A=IntermediatePopulation(Selected_Chromo_Index_A,:);
      chromo_B=IntermediatePopulation(Selected_Chromo_Index_B,:);

      % CROSSOVER
      if (rand < CROSSOVER_RATE)

[child_A,child_B]=Crossover(chromo_A,chromo_B,NUMBER_OF_CROSSOVER_POIN
TS,DISTRIBUTION);
      else
         child_A=chromo_A;
         child_B=chromo_B;
      end

      % MUTATION

mutant_child_A=Mutate(child_A,MUTATION_RATE,MUTATIONS_PER_CHROMOSOME,G
ENE_LENGTH,CHROMO_LENGTH);

mutant_child_B=Mutate(child_B,MUTATION_RATE,MUTATIONS_PER_CHROMOSOME,G
ENE_LENGTH,CHROMO_LENGTH);

      % REPLACE SELECTED CHROMOSOMES WITH CROSSEDOVER AND MUTATED
CHROMOSOMES
      %    replacing provides greater efficiency compared to other GAs
that
      %    require sorting by fitness each generation

IntermediatePopulation(Selected_Chromo_Index_A,:)=mutant_child_A;

IntermediatePopulation(Selected_Chromo_Index_B,:)=mutant_child_B;

      i=i+1; % genetic algorithm works on pairs
   end

    % KEEP BEST CHROMOSOME FROM CURRENT GENERATION
   Old_Best_Chromo=population(1,:);

[gene1dec,gene2dec,gene3dec,gene1bin,gene2bin,gene3bin]=ParseBits(Old_
Best_Chromo,GENE_LENGTH);

Old_Best_Mag=CostFirstRandomDelay([gene1dec;gene2dec;gene3dec],random_
delay);

   % INTERMEDIATE POPULATION IS NOW THE NEW POPULATION
   population=IntermediatePopulation;

   % CALCULATE NEW FITNESS
```

```
[Fitness_Array]=CalcFitnessRandomDelay(population,GENE_LENGTH,random_d
elay);

[Min_Fitness_Mag,Min_Fitness_Index,Mean_Fitness]=FitnessStats(Fitness_
Array);

   % KEEP THE FITEST AND PUT BACK TO TOP!
   % first test which chromosome is best -> new or old?
   if (Min_Fitness_Mag < Old_Best_Mag)
      New_Best_Chromo=population(Min_Fitness_Index,:);
   else
      New_Best_Chromo=Old_Best_Chromo;
   end
   % put best chromosome at top of population
   population(1,:)=New_Best_Chromo;

   % BELOW FOR TESTING ONLY

[gene1dec,gene2dec,gene3dec,gene1bin,gene2bin,gene3bin]=ParseBits(New_
Best_Chromo,GENE_LENGTH);

New_Best_Mag(Generations)=CostFirstRandomDelay([gene1dec;gene2dec;gene
3dec],random_delay);
   % TESTING ONLY FOR ABOVE

    % STORE FITNESS STATISTICS
   Min_array(Generations)=Min_Fitness_Mag;  % store pop fitness
statistics to
   Mean_array(Generations)=Mean_Fitness;       % use for convergence
testing
   Fitness_Matrix=[Fitness_Matrix;Fitness_Array];

    % TEST FOR CONVERGENCE
   %    test for increasing trend in fitness scores
   if (Min_Fitness_Mag == min(Min_array))
      SameMin=SameMin+1;
   else
      SameMin=0;
   end
   if (Min_Fitness_Mag < min(Min_array))
      SmallerMin=TRUE;
   else
      SmallerMin=FALSE;
   end
   if (Mean_Fitness < mean(Mean_array))
      SmallerMean=TRUE;
    else
      SmallerMean=FALSE;
   end


   if ((SmallerMin==TRUE) | (SmallerMean==TRUE) | (SameMin<SAME_MIN))
      Stop = FALSE; % improvement in population fitness therefore
continue
      % fprintf('CONTINUE GA\n');
      % display note to user every 100 generations just to show that
      % GA is still working
      remainder=mod(Generations,100);
      if remainder==0
          if (COMMENTS == TRUE)
            fprintf('No convergence after %d generations, please
wait\n',Generations);
```

```
          end
      end
   else
      Stop = TRUE;
        if (COMMENTS == TRUE)
            fprintf('STOP GA\n');
            fprintf('Generation %d \n\n',Generations);
        end
   end

   if (COMMENTS == TRUE)
       fprintf('Generation %d \n\n',Generations-1);
   end
   Generations=Generations+1; % increment counter of generations
end
```

Annex C to Appendix B *Argo* – InitialisePopulation

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE4111/2
%   Student No:     Q9323707
%
%   InitialisePopulation.m initialises the population of
%   chromsomes by selecting random values for q0,q1 and q2.
%
%   INPUT       pop_size:   the total number of chromsomes
%                           within the population
%               GENE_LENGTH: the constant that defines the number
%                           of bits in a gene
%
%   OUTPUT      Init_Population: an array of randomised
%                           chromosomes in binary string format
%                           whereby the array size is [pop_size,93]
%
%   Updated 18 Sep 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

function [Init_Population]=InitialisePopulation(pop_size,GENE_LENGTH)

% Initialise Seeded_Population matrix
Seeded_Population=zeros(3,pop_size);  % 3 rows for q values
Converted_Population='';

% Populate Seeded_Population matrix
for i=1:3
   for j=1:pop_size
      Seeded_Population(i,j)=(rand*(10737.41823))*100000;
   end
end
Seeded_Population=floor(Seeded_Population);

% Convert decimal to binary string
for i=1:pop_size
   % convert all gene1
   first=Seeded_Population(1,i);
   first=dec2bin(first,(GENE_LENGTH+1)); % leave room for sign bit
   first=num2str(first);
   % convert all gene2
   second=Seeded_Population(2,i);
   second=dec2bin(second,(GENE_LENGTH+1)); % leave room for sign bit
   second=num2str(second);
   % convert all gene3
   third=Seeded_Population(2,i);
   third=dec2bin(third,(GENE_LENGTH+1)); % leave room for sign bit
   third=num2str(third);

   chromo=strcat(first,second,third); % creat chromosome from 3 genes

    Converted_Population=strvcat(Converted_Population,chromo);
    % add the chromosome to the population
```

```
end

% Randomly set first bit i.e. the sign bit
for i=1:pop_size
   sign=round(rand);
   if (sign == 1) % positive
      Converted_Population(i,1)='1';
   else % negative
      Converted_Population(i,1)='0';
   end
   sign=round(rand);
   if (sign == 1) % positive
      Converted_Population(i,32)='1';
   else % negative
      Converted_Population(i,32)='0';
   end
   sign=round(rand);
   if (sign == 1) % positive
      Converted_Population(i,63)='1';
   else % negative
      Converted_Population(i,63)='0';
   end
end

Init_Population=Converted_Population;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Structured English
%
%   seed initial population based on:
%       (i,j)=(rand*(10737.41823))*100000
%   ensure all numbers are rounded down
%
%   convert decimal to binary string:
%   loop for each three componenets (genes) of the seeded matrix
%       convert using dec2str
%       convert using num2str
%       concatenate string of three genes into a chromosome
%       concatenate each chromosome to others to form population
%   end
%
%   randomly set each sign bit of each gene (bits 1,32,3)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%
```

Annex D to Appendix B *Argo* – TournamentSelection

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE4111/2
%   Student No:     Q9323707
%
%   TournamentSelection.m selects a chromosome from the population
%   using "Tournament" selection. That is, it selects at random
%   3 candidate parents and the candidate with the best
%   fitness (i.e. least cost) survives to be a parent.
%
%   INPUT   Fitness_Array:  array of fitness values
%           POP_SIZE:       constant defining the size of population
%
%   OUTPUT  parent:         index into population of the
%                           chromosome selected to be a parent
%
%   Updated 17 Sep 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

function [parent]=TournamentSelection(POP_SIZE,Fitness_Array)

%initialise variables
candidate_parent_1_index=1;
candidate_parent_2_index=1;
candidate_parent_3_index=1;
parent=0;
FALSE=0;
TRUE=1;
same=TRUE;

% select 3 random parent candidates from population,
% repeat random selection if any of the three parents are
% the same
while (same == TRUE)
   candidate_parent_1_index=ceil(rand*POP_SIZE);
    candidate_parent_2_index=ceil(rand*POP_SIZE);
   candidate_parent_3_index=ceil(rand*POP_SIZE);
   same=FALSE;
   if (candidate_parent_1_index == candidate_parent_2_index)
      same = TRUE;
   end
   if (candidate_parent_1_index == candidate_parent_3_index)
      same = TRUE;
   end
   if (candidate_parent_2_index == candidate_parent_3_index)
      same = TRUE;
   end
end

% calculate fitness of the 3 parent candidates
[candidate_parent_1_fitness]=Fitness_Array(candidate_parent_1_index);
[candidate_parent_2_fitness]=Fitness_Array(candidate_parent_2_index);
[candidate_parent_3_fitness]=Fitness_Array(candidate_parent_3_index);
```

```
% determine fitest candidate parent
a=[candidate_parent_1_fitness,candidate_parent_2_fitness,candidate_par
ent_3_fitness];
[minimum,index]=min(a);

if (index == 1)
   parent=candidate_parent_1_index;
end
if (index == 2)
   parent=candidate_parent_2_index;
end
if (index == 3)
   parent=candidate_parent_3_index;
end
```

Annex E to Appendix B *Argo* – ParseBits

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE4111/2
%   Student No:     Q9323707
%
%   ParseBits.m will convert the binary chromosome string into an
%   array of genes to be returned in decimal format
%
%   INPUT   chromosome: string (e.g. '0010101010101010101010101010')
%           GENE_LENGTH: the constant defining the number of
%                       bits in a gene
%
%   OUTPUT  gene1:      decimal (e.g. 1023.1023 )
%           gene2:      decimal (e.g. -234.333 )
%           gene3:      decimal (e.g. 234.333 )
%           gene1:      binary (e.g. '0010101010101010101010101010')
%           gene2:      binary (e.g. '0010101010101010101010101010')
%           gene3:      binary (e.g. '0010101010101010101010101010')
%
%   Updated 18 Sep 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

function
[gene1dec,gene2dec,gene3dec,gene1bin,gene2bin,gene3bin]=ParseBits(chro
mosome,GENE_LENGTH)

% remember chromosome is 93 bits long
% comprised of 3 x (1 + 30) bit encoded gene numbers
% in the format (+/-)xxxxx(.)xxxxx
% provided values between +10737.41823 < q < -10737.41823

% determine the sign of the gene
sign1=chromosome(1);
sign2=chromosome(32);
sign3=chromosome(63);

% get gene values in binary string format
gene1=chromosome((2+0*GENE_LENGTH):(1+1*GENE_LENGTH));
gene2=chromosome((3+1*GENE_LENGTH):(2+2*GENE_LENGTH));
gene3=chromosome((4+2*GENE_LENGTH):(3+3*GENE_LENGTH));

% return the binary string format values for each gene
gene1bin=gene1;
gene2bin=gene2;
gene3bin=gene3;

% convert binary string value to decimal value
gene1=bin2dec(gene1);
gene2=bin2dec(gene2);
gene3=bin2dec(gene3);

% divide by 100,000 to get decimal accuracy
gene1=gene1/100000;
```

```
gene2=gene2/100000;
gene3=gene3/100000;

% correct the sign of the gene decimal and binary values
if sign1 == '0'
   gene1dec=gene1*(-1);
   gene1bin=strcat('0',gene1bin);
else
   gene1bin=strcat('1',gene1bin);
   gene1dec=gene1;
end

if sign2 == '0'
   gene2dec=gene2*(-1);
   gene2bin=strcat('0',gene2bin);
else
   gene2bin=strcat('1',gene2bin);
   gene2dec=gene2;
end

if sign3 == '0'
   gene3dec=gene3*(-1);
   gene3bin=strcat('0',gene3bin);
else
   gene3bin=strcat('1',gene3bin);
   gene3dec=gene3;
end
```

Annex F to Appendix B *Argo* – RouletteWheel

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE4111/2
%   Student No:     Q9323707
%
%   RouletteWheel.m selects a chromosome from the population
%   using "Roulette Wheel" selection. That is, a chromosome
%   fitness is propotional to its chance of selection.
%
%   INPUT   Fitness_Array:  array of fitness values
%           POP_SIZE:       constant defining the number of
%                           chromosomes in a population
%   OUTPUT  Selected_Chromo_Index:  index into the population
%                           array that points to the selected
%                           chromosome
%
%   Updated 17 Sep 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

function [Selected_Chromo_Index]=RouletteWheel(POP_SIZE,Fitness_Array)

% Calculate total fitness value
Total_Fitness=sum(abs(Fitness_Array)); % In terms of probability,
                                       % absolute values only

% Assign each chromosome a prob based on fitness score
Prob_Array=abs(Fitness_Array)./Total_Fitness; % To ensure total 
probability
                                              % equals one, absolute 
values
                                              % only
Cum_Prob_Array=cumsum(Prob_Array); % Create a cummulative probability 
array

% Check that the total probability = 1
Total_Prob=sum(Prob_Array);
error=1-Total_Prob;
if (abs(error) > eps)
   fprintf('ERROR: Total Probability does NOT equal 1, but 
%f\n',Total_Prob);
end

% Assign the selection point ('slice') randomly
Selection_Point=rand;

% Find the chromosome in the population chosen through
% Roulette Wheel selection
Index=0;
for i=1:POP_SIZE
   if ( Selection_Point < Cum_Prob_Array(i) )
      Index=i;
      break;
   end
```

```
end

if (Index == 0) % index not set above
   Index = POP_SIZE; % manually set index to last index
end

Selected_Chromo_Index=Index;
```

Annex G to
Appendix B
***Argo* – Mutate**

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE4111/2
%   Student No:     Q9323707
%
%   Mutate.m mutates a chromosome's bits dependent upon the
%   specified mutation rate.
%
%   INPUT   chromosome:     binary string 93 bits long (e.g.
'0010101')
%                           defining the chromosome that may
%                           or may not become mutated
%           mutation_rate:  decimal value between 1 and 0
%                           defining the liklihood of mutation
%           mutation_number: the maximum number of mutations
%                           allowed per chromosome, either 1 or 2
%
%   OUTPUT  Mutated_Chromosome: binary string 93 bits long
%                           (e.g. '0010101') with individual
%                           bits possibly inverted (mutated)
%                           dependent upon mutation_rate
%
%   Updated 18 Sep 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

function
Mutated_Chromosome=Mutate(chromosome,mutation_rate,mutation_number,GEN
E_LENGTH,CHROMO_LENGTH)

% remember chromosome is 93 bits long
% comprised of 3 x (1 + 30) bit encoded gene numbers
% in the format (+/-)xxxxx(.)xxxxx
% provided values between +10737.41823 < q < -10737.41823

counter=0;

% mutate chromosome
for i=1:(CHROMO_LENGTH)
   if ((rand < mutation_rate) & (i ~= 1) & (i ~= 31) & (i ~= 62))
      % ensure the sign bit is not mutated as this will
      % drastically change the solution
      if chromosome(i) == '1'
         chromosome(i) = '0';
         counter=counter+1;
         %fprintf('MUTATION %d counter is %d mutation_number is
%d\n',i,counter,mutation_number);
         if counter == mutation_number
            break; % only allow mutation_number mutated allele per
chromosome
         end
      elseif chromosome(i) == '0'
         chromosome(i) = '1';
         counter=counter+1;
         %fprintf('MUTATION %d counter is %d mutation_number is
%d\n',i,counter,mutation_number);
         if counter == mutation_number
            break; % only allow mutation_number mutated allele per
chromosome
         end
      end
   end
end

Mutated_Chromosome=chromosome;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Structured English
%
%  loop for chromosome size
%       test if rand less than mutation_rate and is not a sign bit
%           invert sign of allele
%           increment counter
%           test if counter is equal to mutation_number
%               break
%           end
%       end
%   end
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%
```

Annex H to Appendix B *Argo* – Crossover

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE4111/2
%   Student No:     Q9323707
%
%   Crossover.m performs single or double point crossover for a
%   chromosome. The algorithm uses either normal or uniform
%   distributions to select random numbers to determine the
%   crossover point.
%
%   INPUT   chromosome_A:   parent binary string 93 bits long
%                           (e.g. '0010101')
%           chromosome_B:   parent binary string 93 bits long
%                           (e.g. '0010101')
%           point:          the number of crossover points as an
%                           integer, either 1 or 2
%           distribution:   random distribution function, either
'normal'
%                           'uniform'
%
%   OUTPUT  Crossed_chromosome_A: child binary string 93 bits long
%                           (e.g. '0010101') with crossover applied
%           Crossed_chromosome_B: child binary string 93 bits long
%                           (e.g. '0010101') with crossover applied
%
%   Updated 18 Nov 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

function
[Crossed_chromosome_A,Crossed_chromosome_B]=Crossover(chromosome_A,chr
omosome_B,point,distribution)

% GENE_LENGTH=30; not required
CHROMO_LENGTH=length(chromosome_A); % should be 93;
cross_point=0;
cross_point_1=0;
cross_point_2=0;

% DEFAULT to single point crossover

if (point == 2) % double point crossover
   %fprintf('double point\n');
   %fprintf('distribution is %s \n',distribution);
   while (cross_point_1<1 | cross_point_1>93 | cross_point_2<1 |
cross_point_2>93)
    if (strcmpi(distribution,'normal')==1)
        % find crossover point based on random number from a normal
distribution
        cross_point_1 = abs(floor(CHROMO_LENGTH*normrnd(.5,.25)));
        cross_point_2 = abs(floor(CHROMO_LENGTH*normrnd(.5,.25)));
    end
    if (strcmpi(distribution,'uniform')==1)
        % find crossover point based on random number from a uniform
distribution
```

```
        cross_point_1 = floor(CHROMO_LENGTH*rand);
         cross_point_2 = floor(CHROMO_LENGTH*rand);
      end
      if (cross_point_1 > cross_point_2) % sort the cross points
         temp = cross_point_1;
         cross_point_1 = cross_point_2;
         cross_point_2 = temp;
      end
   end
   if ((cross_point_1 == 1) | (cross_point_1 == 32) | (cross_point_1
== 63))
      cross_point_1 = cross_point_1 +1; % move crossover point off the
sign value
   end
   if ((cross_point_2 == 1) | (cross_point_2 == 32) | (cross_point_2
== 63))
      cross_point_2 = cross_point_2 +1; % move crossover point off the
sign value
   end
   %fprintf('crossover point one is %d crossover point two is
%d\n',cross_point_1,cross_point_2);
   % perform first point crossover on chromosomes
   prefixA=chromosome_A(1:cross_point_1);
   prefixB=chromosome_B(1:cross_point_1);
   postfixA=chromosome_A((cross_point_1+1):CHROMO_LENGTH);
   postfixB=chromosome_B((cross_point_1+1):CHROMO_LENGTH);
   Crossed_chromosome_A=strcat(prefixA,postfixB);
   Crossed_chromosome_B=strcat(prefixB,postfixA);

   % perform second point crossover on chromosomes
   prefixA=Crossed_chromosome_A(1:cross_point_2);
   prefixB=Crossed_chromosome_B(1:cross_point_2);
   postfixA=Crossed_chromosome_A((cross_point_2+1):CHROMO_LENGTH);
   postfixB=Crossed_chromosome_B((cross_point_2+1):CHROMO_LENGTH);
   Crossed_chromosome_A=strcat(prefixA,postfixB);
   Crossed_chromosome_B=strcat(prefixB,postfixA);
   return;
end

if (point == 1) % single point crossover
   %fprintf('single point\n');
   %fprintf('distribution is %s \n',distribution);
   while (cross_point<1 | cross_point >93)
      if (strcmpi(distribution,'normal')==1)
        % find crossover point based on random number from a normal
distribution
        cross_point = abs(floor(CHROMO_LENGTH*normrnd(.5,.25)));
    end
    if (strcmpi(distribution,'uniform')==1)
        % find crossover point based on random number from a uniform
distribution
        cross_point = floor(CHROMO_LENGTH*rand);
    end
    if ((cross_point == 1) | (cross_point == 32) | (cross_point ==
63))
        cross_point = cross_point +1; % move crossover point off the
sign value
      end
   end
   % fprintf('crossover point is %d\n',cross_point);
   % perform single point crossover on chromosomes
   prefixA=chromosome_A(1:cross_point);
   prefixB=chromosome_B(1:cross_point);
   postfixA=chromosome_A((cross_point+1):CHROMO_LENGTH);
   postfixB=chromosome_B((cross_point+1):CHROMO_LENGTH);
   Crossed_chromosome_A=strcat(prefixA,postfixB);
   Crossed_chromosome_B=strcat(prefixB,postfixA);
   return;
end
```

Annex I to Appendix B *Argo* – CalcFitnessRandomDelay

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE4111/2
%   Student No:     Q9323707
%
%   CalcFitnessRandomDelay.m calculates the fitness
%   of the population using the CostFirstRandomDelay.m function.
%
%   INPUT   population:     population of chromosomes in binary
%                           string format
%           GENE_LENGTH:    constant defining the number of bits in a
%                           gene
%           random_delay:   the random delay to the start of the input
%                           signal
%
%   OUTPUT  Fitness_Array:  decimal array of fitness values for
%                           the entire population
%
%   Updated 18 Sep 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

function
[Fitness_Array]=CalcFitnessRandomDelay(population,GENE_LENGTH,random_d
elay)

pop_size=size(population,1);

for i=1:pop_size
    chromosome=population(i,:);

[gene1dec,gene2dec,gene3dec,gene1bin,gene2bin,gene3bin]=ParseBits(chro
mosome,GENE_LENGTH)
   % replace CostFirstRandomDelay with CostSecond for second order
system

Fitness_Array(i)=(CostFirstRandomDelay([gene1dec;gene2dec;gene3dec],ra
ndom_delay));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Structured English
%
%  loop for population size
%       assign population(i,:) to chromosome
%       ParseBits
%       assign cost to Fitness_Array(i)
%   end
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%
```

Annex J to Appendix B *Argo* – CostFirst

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE3105
%   Student No: Q9323707
%
%   CostFirst.m calculates the value of S (cost function)
%   for the First Order Test Control system given the input values
%   of q0, q1, and q2 for the PID controller.
%
%   The cost function, S is calculated using the ITAE criterion:
%
%       S = Summation of (k*|e(k)|) a total of M times
%
%   INPUT   Q:              is an array of values representing the PID
%                           parameters q0, q1 and q2
%
%   OUTPUT  error:          the final error value
%           C:              is an array of values for c(k) - output
signal
%           E:              is an array of values for e(k) - error
signal
%
%   Updated 18 Nov 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

function [error,C,E]=CostFirst(Q)
Q0=Q(1);
Q1=Q(2);
Q2=Q(3);

% m is set for 100 summations for this exercise
m=100;
% T assigned 0.1 in order to simplify calculations
T=0.1;

a=-exp(-0.5);

E=zeros(m,1); %zeroise error array
C=zeros(m,1); %zeroise output array
R=zeros(m,1); %zeroise input array

for k=9:m
    R(k)=1; %step input delayed after 9 samples i.e. 9*.1 sec = .9 sec
delay to input
end

%calculate error and output arrays
for k=9:m
   r=R(k)-(1-a)*R(k-1)-a*R(k-2);
   e=(1-a)*E(k-1)+a*E(k-2)-Q0*(1+a)*E(k-6)-Q1*(1+a)*E(k-7)-
Q2*(1+a)*E(k-8);
   E(k)=r+e;
   C(k)=(1-a)*C(k-1)+a*C(k-2)+Q0*(1+a)*E(k-6)+Q1*(1+a)*E(k-
7)+Q2*(1+a)*E(k-8);
```

```
end

%calculate error
error=0;
i=1:m;
k_error=i.*abs(transpose(E));
error=sum(k_error);
```

Annex K to Appendix B *Argo* – CostFirstRandomDelay

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE3105
%   Student No:     Q9323707
%
%   CostFirstRandomDelay.m calculates the value of S (cost function)
%   for the First Order Test Control system given the input values
%   of q0, q1, and q2 for the PID controller.
%
%   The cost function, S is calculated using the ITAE criterion:
%
%       S = Summation of (k*|e(k)|) a total of M times
%
%   INPUT   Q:              is an array of values representing the PID
%                           parameters q0, q1 and q2
%           random_delay:   is the random delay to the start of the
%                           input signal
%   OUTPUT  error:          the final error value
%           C:              is an array of values for c(k) - output
signal
%           E:              is an array of values for e(k) - error
signal
%
%   Updated 18 Nov 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

function [error,C,E]=CostFirstRandomDelay(Q,random_delay)
Q0=Q(1);
Q1=Q(2);
Q2=Q(3);
%Q3=random_delay;

% m is set for 100 summations for this exercise
m=100;
% T assigned 0.1 in order to simplify calculations
T=0.1;
random_delay=abs(floor(random_delay));  %ensure a positive delay and
sample number an integer
a=-exp(-0.5);

E=ones(m,1); %zeroise error array to be unit input signal until delay
C=zeros(m,1); %zeroise output array
R=zeros(m,1); %zeroise input array

for k=(random_delay+3):m
    R(k)=1; %step input delayed after random delay
end

%calculate error and output arrays
for k=(random_delay+3):m
   r=R(k)-(1-a)*R(k-1)-a*R(k-2);
   e=(1-a)*E(k-1)+a*E(k-2)-Q0*(1+a)*E(k-random_delay)-Q1*(1+a)*E(k-
random_delay-1)-Q2*(1+a)*E(k-random_delay-2);
```

```
   E(k)=r+e;
   %   calculation of output value not required, only error for the 
purposes
   %   of calculating the cost function
   C(k)=(1-a)*C(k-1)+a*C(k-2)+Q0*(1+a)*E(k-random_delay)+Q1*(1+a)*E(k-
random_delay-1)+Q2*(1+a)*E(k-random_delay-2);
end

%calculate error ITAE
error=0;
i=1:m;
k_error=i.*abs(transpose(E));
error=sum(k_error);
```

Annex L to Appendix B *Argo* – CostSecond

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Name:           Matthew Mackenzie
%   Unit:           ELE3105
%   Student No:     Q9323707
%
%   CostSecond.m calculates the value of S (cost function)
%   for the Second Order Test Control system given the input values
%   of q0, q1, and q2 for the PID controller.
%
%   The cost function, S is calculated using the ITAE criterion:
%
%       S = Summation of (k*|e(k)|) a total of M times
%
%   INPUT   Q:              is an array of values representing the PID
%                           parameters q0, q1 and q2
%
%   OUTPUT  error:          the final error value
%           C:              is an array of values for c(k) - output
signal
%           E:              is an array of values for e(k) - error
signal
%
%   Updated 18 Nov 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%

function [error,C,E]=CostSecond(Q)
q0=Q(1);
q1=Q(2);
q2=Q(3);

% m is set for 100 summations for this exercise
m=101;
% T was calculated in a previous assignment part
T=0.05;

% set up simultaneous equations to solve for AX=B where X=[e;c;m]
a=(3/109);
b=(1/109)*(-3*exp(-3*T)*cos(10*T)+10*exp(-3*T)*sin(10*T));
c=(1/109)*(3*exp(-6*T)-3*exp(-3*T)*cos(10*T)-10*exp(-3*T)*sin(10*T));
d=-2*exp(-3*T)*cos(10*T);
e=exp(-6*T);

A=[1 1 0;-q0 0 1;0 1 -a];

%first iteration
B=[0;0;0];
X=inv(A)*B;
E(1)=X(1);
C(1)=X(2);
M(1)=X(3);

%second iteration
B=[1;0;0];
```

```
X=inv(A)*B;
E(2)=X(1);
C(2)=X(2);
M(2)=X(3);

%iterative after first two
for k=3:(m+1)
   b1=k-1;  % r(k) = 1 for unit ramp input, for k > 0
    b2=q1*E(k-1)+q2*E(k-2)+M(k-1);
    b3=b*M(k-1)+c*M(k-2)-d*C(k-1)-e*C(k-2);
    B=[b1;b2;b3];
    X=inv(A)*B; % the \ operator is more efficient than the matlab
function inv() for this application
    E(k)=X(1);
    C(k)=X(2);
   M(k)=X(3);
end

%calculate error
error=0;
%for i=1:100
i=1:(m+1);
k_error=(i-1).*abs(E);
error=sum(k_error);
%end
```

Annex M to Appendix B *Argo* – ARGOGUI

```
function ARGOGUI(fcn)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This GUI code is based on the source code downloaded from
%   Cornell University
%
%   Course:     BioNB 441 - Biological Instrumentation
%   Subject:    Graphical User Interface Design
%   Staff:      Bruce Land
%   Date:       May 2007
%   Address:
%   http://courses.cit.cornell.edu/bionb442/GUIdesign/GUIdemo.m
%
% The GUI code has then been extensively modified by Matthew Mackenzie
%   as part of ENG4111/112 Reasearch Project
%   25 Sep 07
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%This code detects the first entry into the function
%from the command line with no parameters
if nargin == 0
   fcn = 'makeGUI';
end

%This is the main decision point of the function.
%The switch statement is executed once-per-fuction call
switch fcn

   %This code is executed ONCE when the function enters with
   %no arguments
case 'makeGUI'

   % Determine the name of this function and store it in the
   %figure plotinfo variable.
   %Since variables used in a function are not persistent after
   %the function exits we will need to store the state-variables
   %in a data structure associated with the persistent Figure-window.
   %The plotinfo sturcture will be saved into the Figure's UserData
   %area and retrieved from there as necessary.
   plotinfo.myname = mfilename;

   % ===Create main figure==========================
   fig = figure('Position',centerfig(900,600),...
      'Resize','off',...
      'NumberTitle','off',...
      'Name','Genetic Algorithm Parameter Input Menu',...
      'Interruptible','off',...
      'Toolbar','figure',...
      'Menubar','figure',...
      'Color',get(0,'DefaultUIControlBackgroundColor'));

   %===Header Text====================================
   uicontrol(gcf,'Style','text', ...
      'String','Genetic Algorithm Parameter Input Menu',...
      'fontsize',18, ...
      'HorizontalAlignment','Center',...
      'Position',[100,565,700,30],...
      'BackgroundColor',[0.4 0.7 0.8]);

   % ===Create Axes==================================
   plotinfo.ax = axes('Units','pixels',...
      'Position',[235 50 580 480],...
      'Box','on',...
      'XLim',[0 1],'YLim',[-1 1]);
   xlabel('Generations'); ylabel('Cost');

   %==Frequency slider===========================
%   plotinfo.freq=1.0;
%   plotinfo.s1 = uicontrol(gcf,'Style','text', ...
%      'String','frequency',...
%      'Position',[10,240,100,20],...
%      'BackgroundColor',[0.8,0.8,0.8]);
%   plotinfo.s2 = uicontrol(gcf,'Style','edit',...
%      'String',num2str(plotinfo.freq),...
%      'Position',[110,240,50,20],...
%      'BackgroundColor',[0.8,0.8,0.8],...
%      'callback', [plotinfo.myname,' editfreq']);
%   plotinfo.s3 = uicontrol(gcf,...
%      'Style','slider',...
%      'Min' ,1,'Max',20, ...
%      'Position',[10,220,150,20], ...
%      'Value', 1,...
%      'SliderStep',[0.01 0.1], ...
%      'BackgroundColor',[0.8,0.8,0.8],...
%      'CallBack', [plotinfo.myname,' setfreq']);

   %==Crossover ===========================
   plotinfo.cross=0.5;
   plotinfo.c1 = uicontrol(gcf,'Style','text', ...
      'String','Crossover Rate',...
      'Position',[10,240,100,20],...
      'BackgroundColor',[0.8,0.8,0.8]);
   plotinfo.c2 = uicontrol(gcf,'Style','edit',...
      'String',num2str(plotinfo.cross),...
      'Position',[110,240,50,20],...
      'BackgroundColor',[0.8,0.8,0.8],...
      'callback', [plotinfo.myname,' editcross']);

   %==Simulations ===========================
   plotinfo.sims=3;
   plotinfo.d1 = uicontrol(gcf,'Style','text', ...
      'String','Simulation Runs',...
      'Position',[10,210,100,20],...
      'BackgroundColor',[0.8,0.8,0.8]);
   plotinfo.d2 = uicontrol(gcf,'Style','edit',...
      'String',num2str(plotinfo.sims),...
      'Position',[110,210,50,20],...
      'BackgroundColor',[0.8,0.8,0.8],...
      'callback', [plotinfo.myname,' editsims']);

   %==Number of Crossover Points ===========================
   plotinfo.crosspt=1;
   plotinfo.e1 = uicontrol(gcf,'Style','text', ...
      'String','# Crossover Points',...
      'Position',[10,270,100,20],...
      'BackgroundColor',[0.8,0.8,0.8]);
   plotinfo.e2 = uicontrol(gcf,'Style','edit',...
      'String',num2str(plotinfo.crosspt),...
      'Position',[110,270,50,20],...
      'BackgroundColor',[0.8,0.8,0.8],...
      'callback', [plotinfo.myname,' editcrosspt']);

   %==Mutation Rate ===========================
   plotinfo.mute=.001;
   plotinfo.f1 = uicontrol(gcf,'Style','text', ...
      'String','Mutation Rate',...
      'Position',[10,300,100,20],...
      'BackgroundColor',[0.8,0.8,0.8]);
   plotinfo.f2 = uicontrol(gcf,'Style','edit',...
      'String',num2str(plotinfo.mute),...
      'Position',[110,300,50,20],...
      'BackgroundColor',[0.8,0.8,0.8],...
      'callback', [plotinfo.myname,' editmute']);

   %==Mutations per Chromo ===========================
   plotinfo.mutenum=1;
   plotinfo.g1 = uicontrol(gcf,'Style','text', ...
      'String','# Mutations',...
      'Position',[10,330,100,20],...
      'BackgroundColor',[0.8,0.8,0.8]);
   plotinfo.g2 = uicontrol(gcf,'Style','edit',...
      'String',num2str(plotinfo.mutenum),...
      'Position',[110,330,50,20],...
      'BackgroundColor',[0.8,0.8,0.8],...
      'callback', [plotinfo.myname,' editmutenum']);

   %==Maximum Generations ===========================
   plotinfo.gen=100;
   plotinfo.h1 = uicontrol(gcf,'Style','text', ...
      'String','Max Generations',...
      'Position',[10,360,100,20],...
      'BackgroundColor',[0.8,0.8,0.8]);
   plotinfo.h2 = uicontrol(gcf,'Style','edit',...
      'String',num2str(plotinfo.gen),...
      'Position',[110,360,50,20],...
      'BackgroundColor',[0.8,0.8,0.8],...
      'callback', [plotinfo.myname,' editgen']);

   %==Population to Keep ===========================
   plotinfo.keep=.7;
   plotinfo.i1 = uicontrol(gcf,'Style','text', ...
      'String','% Keep',...
      'Position',[10,390,100,20],...
      'BackgroundColor',[0.8,0.8,0.8]);
   plotinfo.i2 = uicontrol(gcf,'Style','edit',...
      'String',num2str(plotinfo.keep),...
      'Position',[110,390,50,20],...
      'BackgroundColor',[0.8,0.8,0.8],...
      'callback', [plotinfo.myname,' editkeep']);

  % ===The Quit Button================================
   uicontrol(gcf,'Style','pushbutton',...
      'String','Quit',...
      'Interruptible','off',...
      'BusyAction','cancel',...
      'Position',[840 20 45 25],...
      'BackgroundColor',[1,0.8,0.8], ...
      'Callback',[plotinfo.myname,' quit']);

  %==The Launch GA===========================
   uicontrol(gcf,'Style','pushbutton', ...
      'String','Launch ARGOR',...
      'Interruptible','off',...
      'BusyAction','cancel',...
      'Position',[10,60,70,20],...
      'BackgroundColor',[1,0.8,0.8], ...
      'CallBack',[plotinfo.myname,' launchga']);

   %==Selection Chooser========================
   plotinfo.sel='Tournament';
   plotinfo.selchoice=uicontrol(gcf,'Style','PopupMenu', ...
      'String','Tournament|Roulette',...
      'Position',[10,430,150,50],...
      'BackgroundColor',[0.8,0.8,0.8], ...
      'CallBack',[plotinfo.myname,' selchoice'] );

   %==Distribution Chooser========================
   plotinfo.dist='uniform';
   plotinfo.distchoice=uicontrol(gcf,'Style','PopupMenu', ...
      'String','uniform|random',...
      'Position',[10,390,150,50],...
      'BackgroundColor',[0.8,0.8,0.8], ...
      'CallBack',[plotinfo.myname,' distchoice'] );

   %==Axes Title====================================
   plotinfo.title='Use edit field to change the plot title';
   plotinfo.ttl=uicontrol(gcf,'Style','edit', ...
      'String','Edit Figure title',...
      'Position',[10,500,150,20],...
      'BackgroundColor',[1,1,1], ...
      'CallBack',[plotinfo.myname,' edttl'] );

   %==Context sensitive menu==========================
   %====Also note reference to this menu in the plot==
   % Define the context menu (taken from Matlab docs)
   plotinfo.cmenu = uicontextmenu;
   % Define the context menu items
    plotinfo.item1 = uimenu(plotinfo.cmenu, 'Label', 'dashed', ...
       'Callback',[plotinfo.myname,' linemenu1']) ;
    plotinfo.item2 = uimenu(plotinfo.cmenu, 'Label', 'dotted', ...
       'Callback', [plotinfo.myname,' linemenu2']);
    plotinfo.item3 = uimenu(plotinfo.cmenu, 'Label', 'solid',...
       'Callback', [plotinfo.myname,' linemenu3']);
    uicontrol('style','text',...
       'string','Hit Lauch ARGO button, then right-click the plot line
for options',...
       'backgroundcolor','white',...
       'position',[100,50,100,60]);

   %put all the variables in a safe place (the figure's data area)
   set(fig,'UserData',plotinfo);

%case 'setfreq'
%   %Get data from the figure's data area
%   plotinfo=get(gcf,'UserData');
%   %Get the value from the slider
%   plotinfo.freq=get(plotinfo.s3,'Value');
%   %Update the text which shows the slider value
%   set(plotinfo.s2,'String',plotinfo.freq);
%   %Store the new slider value back into the figure's data area
%   set(gcf,'UserData',plotinfo);
```

```
%case 'editfreq'
%   plotinfo=get(gcf,'UserData');
%   plotinfo.freq=str2num(get(plotinfo.s2,'string'));
%   set(plotinfo.s3,'value',plotinfo.freq);
%   set(gcf,'UserData',plotinfo);

case 'editcross'
   plotinfo=get(gcf,'UserData');
   plotinfo.cross=str2num(get(plotinfo.c2,'string'));
  % set(plotinfo.c3,'value',plotinfo.cross);
   set(gcf,'UserData',plotinfo);

case 'editsims'
   plotinfo=get(gcf,'UserData');
   plotinfo.sims=str2num(get(plotinfo.d2,'string'));
  % set(plotinfo.c3,'value',plotinfo.cross);
   set(gcf,'UserData',plotinfo);

case 'editcrosspt'
   plotinfo=get(gcf,'UserData');
   plotinfo.crosspt=str2num(get(plotinfo.e2,'string'));
  % set(plotinfo.c3,'value',plotinfo.cross);
   set(gcf,'UserData',plotinfo);

case 'editmute'
   plotinfo=get(gcf,'UserData');
   plotinfo.mute=str2num(get(plotinfo.f2,'string'))
  % set(plotinfo.c3,'value',plotinfo.cross);
   set(gcf,'UserData',plotinfo);

case 'editmutenum'
   plotinfo=get(gcf,'UserData');
   plotinfo.mutenum=str2num(get(plotinfo.g2,'string'))
  % set(plotinfo.c3,'value',plotinfo.cross);
   set(gcf,'UserData',plotinfo);

case 'editgen'
   plotinfo=get(gcf,'UserData');
   plotinfo.gen=str2num(get(plotinfo.h2,'string'))
  % set(plotinfo.c3,'value',plotinfo.cross);
   set(gcf,'UserData',plotinfo);

case 'editkeep'
   plotinfo=get(gcf,'UserData');
   plotinfo.keep=str2num(get(plotinfo.i2,'string'))
  % set(plotinfo.c3,'value',plotinfo.cross);
   set(gcf,'UserData',plotinfo);

case 'selchoice'
   plotinfo=get(gcf,'UserData');
   plotinfo.sel=get(plotinfo.selchoice,'value');
   switch plotinfo.sel
   case 1
      plotinfo.sel='Tournament';
   case 2
      plotinfo.sel='Roulette';
   end
   set(gcf,'UserData',plotinfo);

case 'distchoice'
   plotinfo=get(gcf,'UserData');
   plotinfo.dist=get(plotinfo.distchoice,'value');
   switch plotinfo.dist
   case 1
      plotinfo.dist='uniform';
   case 2
      plotinfo.dist='random';
   end
   set(gcf,'UserData',plotinfo);

case 'edttl'
   plotinfo=get(gcf,'UserData');
   plotinfo.title=get(plotinfo.ttl,'string');
   title(plotinfo.title);
   set(gcf,'UserData',plotinfo);

case 'linemenu1'
   plotinfo=get(gcf,'UserData');
   set(plotinfo.line, 'LineStyle', '--')

case 'linemenu2'
   plotinfo=get(gcf,'UserData');
   set(plotinfo.line, 'LineStyle', ':')

case 'linemenu3'
   plotinfo=get(gcf,'UserData');
   set(plotinfo.line, 'LineStyle', '-')

case 'launchga'
    plotinfo=get(gcf,'UserData');
%   number of sims      = plotinfo.sims
%   max generations     = plotinfo.gen
%   selection method    = plotinfo.sel
%   distribution type   = plotinfo.dist
%   population kept     = plotinfo.keep
%   mutation rate       = plotinfo.mute
%   # mutations         = plotinfo.mutenum
%   crossover rate      = plotinfo.cross
%   number of crossovers= plotinfo.crosspt
   launch_reply = questdlg('Execute ARGO?');
   if strcmp(launch_reply,'Yes')
      fig=gcf;
      refresh(fig);
      tic
      ARGOR(    plotinfo.sims,...
                plotinfo.gen,...
                plotinfo.sel,...
                plotinfo.dist,...
                plotinfo.keep,...
                plotinfo.mute,...
                plotinfo.mutenum,...
                plotinfo.cross,...
                plotinfo.crosspt);
       duration=toc;
       duration=num2str(duration);
       message1='Processing time was  ';
       message2=' seconds';
       message=strcat(message1,duration,message2);
       msgbox(message);
   end

case 'quit'
   fig = gcf;
   quit_reply = questdlg('Are you sure you wish to quit ARGO?');
   if strcmp(quit_reply,'Yes')
```

```
      close(fig);
   end


end

%===A utility to center the window on the screen============
function pos = centerfig(width,height)

% Find the screen size in pixels
screen_s = get(0,'ScreenSize');
pos = [screen_s(3)/2 - width/2, screen_s(4)/2 - height/2, width,
height];
```

Appendix C

Cost Functions

This project uses three cost functions in order to test the system:

a) `CostFirst` – models a first order control system,

b) `CostSecond` – models a second order system, and

c) `CostFirstRandomDelay` – models the same first order control system but with a random delayed input.

This appendix provides the mathematical background for calculating the respective costs of the test control systems.

For ease of reading, the first order system was modelled by direct solution of equations for $E(Z), C(Z)$ and $M(Z)$; whereas the second order system was modelled by solving a set of simultaneous equations for $E(Z), C(Z)$ and $M(Z)$.

The solution for the second order system was conducted as part of this project; whereas, the solution for the second order system was conducted as part of a previous assignment for ELE3105 Computer Controlled Systems at the University of Southern Queensland (Mackenzie, 2004).

C.1 First Order System

The `CostFirst` function was calculated from the transfer function $G_p(s)$, such that:

Given,

$$G_p(s) = \frac{5e^{-5Ts}}{s+5}$$

EQN 1

Then,

$$\begin{aligned} G_{HP}(z) &= \left(1 - z^{-1}\right) \times Z\left\{\frac{G_P(s)}{s}\right\} \\ &= \left(1 - z^{-1}\right) \times Z\left\{\frac{\frac{5e^{-5Ts}}{s+5}}{s}\right\} \end{aligned}$$

EQN 2

Let,

$$T = 0.1$$

EQN 3

Then by definition,

$$\begin{aligned} z &= e^{sT} \\ z^{-5} &= e^{-(5 \times 0.1 \times s)} \\ z^{-5} &= e^{-0.5s} \end{aligned}$$

EQN 4

Then substituting EQN 4 back into EQN 2, $G_{HP}(z)$ simplifies to,

$$\begin{aligned} G_{HP}(z) &= \left(1 - z^{-1}\right)z^{-5} \times Z\left\{\frac{\frac{5}{s+5}}{s}\right\} \\ &= \left(1 - z^{-1}\right)z^{-5} \times Z\left\{\frac{5}{s(s+5)}\right\} \end{aligned}$$

EQN 5

Then using Partial Fractions to simplify the Z Transform,

$$\frac{5}{s(s+5)} = \frac{A}{s} + \frac{B}{(s+5)}$$
$$= \frac{As + 5A + Bs}{s(s+5)}$$

EQN 6

Comparing coefficients,

$$s^0 : 5 = 5A$$
$$A = 1$$
$$s^1 : 0 = A + B$$
$$B = -1$$

EQN 7

Then substituting the partial fraction equivalent back into EQN 5, it simplifies to,

$$G_{HP}(z) = (1 - z^{-1})z^{-5} \times Z\left\{\frac{\frac{5}{s+5}}{s}\right\}$$
$$= (1 - z^{-1})z^{-5} \times Z\left\{\frac{5}{s(s+5)}\right\}$$
$$= z^{-5}(1 - z^{-1}) \times \left\{\frac{1}{(1 - z^{-1})} - \frac{1}{(1 - e^{-5T}z^{-1})}\right\}$$
$$= z^{-5}(1 - z^{-1}) \times \left\{\frac{(1 - e^{-5T}z^{-1}) - (1 - z^{-1})}{(1 - z^{-1})(1 - e^{-5T}z^{-1})}\right\}$$
$$= z^{-5} \times \left\{\frac{(z^{-1} - e^{-5T}z^{-1})}{(1 - e^{-5T}z^{-1})}\right\}$$
$$= z^{-6} \times \left\{\frac{(1 - e^{-5T})}{(1 - e^{-5T}z^{-1})}\right\} \qquad remembering\ T = 0.1$$
$$= z^{-6} \times \left\{\frac{(1 - e^{-0.5})}{(1 - e^{-0.5}z^{-1})}\right\}$$

EQN 8

Now that $G_{HP}(z)$ has been found, need to solve the following set of equations,

$$E(Z) = R(Z) - C(Z)$$

EQN 9

$$C(Z) = G_{HP}(Z) \times M(Z)$$

EQN 10

$$M(Z) = G_C(Z) \times E(Z)$$

EQN 11

Substituting EQN 10 into EQN 9 results in,

$$E(Z) = R(Z) - G_{HP}(Z) \times M(Z)$$

EQN 12

Then substituting EQN 11 into EQN 12 results in,

$$\begin{aligned} E(Z) &= R(Z) - G_{HP}(Z) \times G_C(Z) \times E(Z) \\ E(Z)\left(1 + G_{HP}(Z) \times G_C(Z)\right) &= R(Z) \\ &= \frac{R(Z)}{\left(1 + G_{HP}(Z) G_C(Z)\right)} \end{aligned}$$

EQN 13

Noting that $G_C(Z)$ for a PID Controller has the standard equation,

$$G_C(Z) = \frac{\left(q_0 + q_1 z^{-1} + q_2 z^{-2}\right)}{\left(1 - z^{-1}\right)}$$

EQN 14

Now EQN 13 can be solved substituting in EQN 8 and EQN 14,

$$\begin{aligned} E(Z) &= \frac{R(Z)}{\left(1 + G_{HP}(Z) G_C(Z)\right)} \\ &= \frac{R(Z)}{\left(1 + \left\{ z^{-6} \times \frac{\left(1 - e^{-0.5}\right)}{\left(1 - e^{-0.5} z^{-1}\right)} \right\} \left\{ \frac{\left(q_0 + q_1 z^{-1} + q_2 z^{-2}\right)}{\left(1 - z^{-1}\right)} \right\}\right)} \\ &= \frac{R(Z)}{\left(1 + \left\{ \frac{q_0 z^{-6} + q_1 z^{-7} + q_2 z^{-8} - e^{-0.5} q_0 z^{-6} - e^{-0.5} q_1 z^{-7} - e^{-0.5} q_2 z^{-8}}{1 - z^{-1} - e^{-0.5} z^{-1} + e^{-0.5} z^{-2}} \right\}\right)} \\ &= \frac{R(Z)}{\left(\frac{1 - z^{-1} - e^{-0.5} z^{-1} + e^{-0.5} z^{-2} + q_0 z^{-6} + q_1 z^{-7} + q_2 z^{-8} - e^{-0.5} q_0 z^{-6} - e^{-0.5} q_1 z^{-7} - e^{-0.5} q_2 z^{-8}}{1 - z^{-1} - e^{-0.5} z^{-1} + e^{-0.5} z^{-2}} \right)} \\ &= R(Z) \left(\frac{1 - z^{-1} - e^{-0.5} z^{-1} + e^{-0.5} z^{-2}}{1 - z^{-1} - e^{-0.5} z^{-1} + e^{-0.5} z^{-2} + q_0 z^{-6} + q_1 z^{-7} + q_2 z^{-8} - e^{-0.5} q_0 z^{-6} - e^{-0.5} q_1 z^{-7} - e^{-0.5} q_2 z^{-8}} \right) \end{aligned}$$

EQN 15

EQN 15 can then be transformed into a difference equation,

$$
\begin{aligned}
E(Z) &= R(Z)\left[\frac{1-z^{-1}-e^{-0.5}z^{-1}+e^{-0.5}z^{-2}}{1-z^{-1}-e^{-0.5}z^{-1}+e^{-0.5}z^{-2}+q_0z^{-6}+q_1z^{-7}+q_2z^{-8}-e^{-0.5}q_0z^{-6}-e^{-0.5}q_1z^{-7}-e^{-0.5}q_2z^{-8}}\right] \\
&= R(Z)\left[\frac{1-(1-a)z^{-1}-az^{-2}}{1-(1-a)z^{-1}-az^{-2}+q_0(1+a)z^{-6}+q_1(1+a)z^{-7}+q_2(1+a)z^{-8}}\right] \\
e_k &= r_k-(1-a)r_{k-1}-ar_{k-2}+(1-a)e_{k-1}+ae_{k-2}-q_0(1+a)e_{k-6}-q_1(1+a)e_{k-7}-q_2(1+a)e_{k-8}
\end{aligned}
$$

EQN 16

Where

$$a = -e^{-0.5}$$

EQN 17

Finally, a difference equation for C(Z) can then be derived. Firstly, substitute EQN 11 into EQN 10,

$$
\begin{aligned}
C(Z) &= G_{HP}(Z)\times M(Z) \\
&= G_{HP}(Z)\times G_C(Z)\times E(Z) \\
&= \left[\left\{z^{-6}\times\frac{\left(1-e^{-0.5}\right)}{\left(1-e^{-0.5}z^{-1}\right)}\right\}\left\{\frac{\left(q_0+q_1z^{-1}+q_2z^{-2}\right)}{\left(1-z^{-1}\right)}\right\}\right]\times E(Z) \\
&= \left[\frac{q_0z^{-6}+q_1z^{-7}+q_2z^{-8}-e^{-0.5}q_0z^{-6}-e^{-0.5}q_1z^{-7}-e^{-0.5}q_2z^{-8}}{1-z^{-1}-e^{-0.5}z^{-1}+e^{-0.5}z^{-2}}\right]\times E(Z) \\
&= E(Z)\left[\frac{q_0(1+a)z^{-6}+q_1(1+a)z^{-7}+q_2(1+a)z^{-8}}{1-(1-a)z^{-1}-az^{-2}}\right] \\
c_k &= (1-a)c_{k-1}+ac_{k-2}+q_0(1+a)e_{k-6}+q_1(1+a)e_{k-7}+q_2(1+a)e_{k-8}
\end{aligned}
$$

EQN 18

Where,

$$a = -e^{-0.5}$$

EQN 19

Now EQN 16 and EQN 19 can then be simulated using a step input. The MATLAB simulation code is found at Annex J to Appendix B ***Argo*** – CostFirst.

The built-in MATLAB® function `fminsearch` indicated that the global minimum was obtained by the parameter values:

$$
\begin{aligned}
q_0 &= 1.23603318967131 \\
q_1 &= -1.73364928005939 \\
q_3 &= 0.634199343789116
\end{aligned}
$$

EQN 20

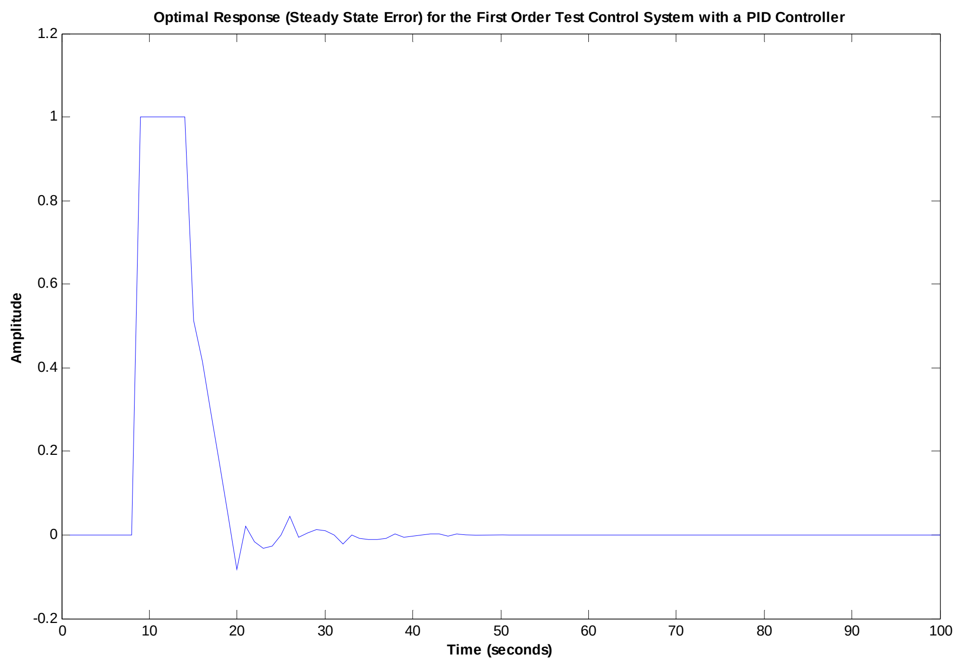


Figure C -1 Optimum Response (Steady State Error) for the First Order Test Control System with a PID Controller with Delayed Start and Unit Step Input

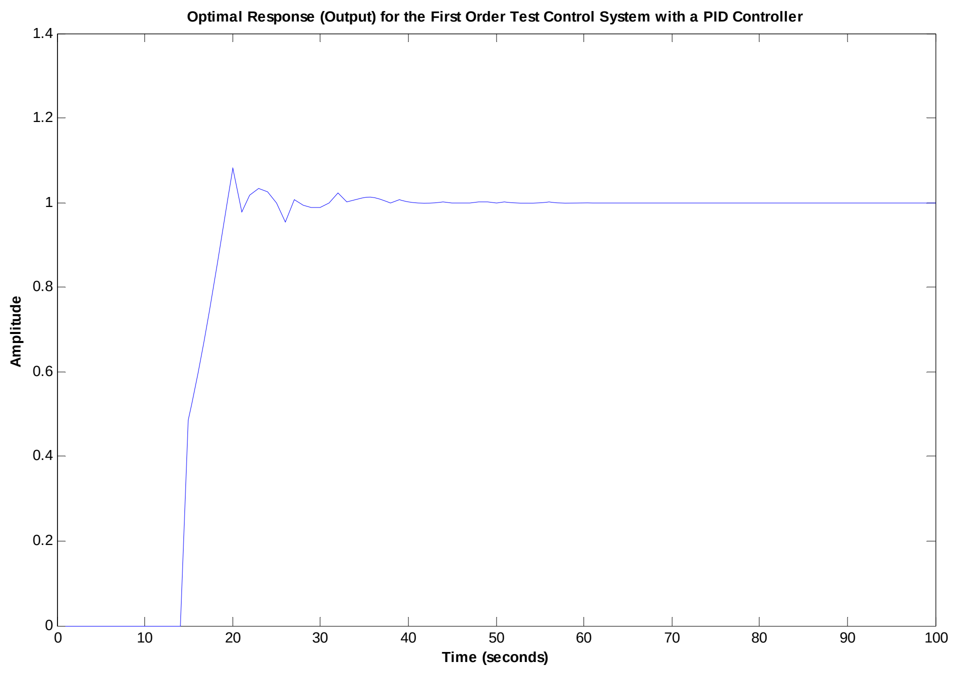


Figure C -2 Optimum Response (Output) for the First Order Test Control System with a PID Controller with Delayed Start and Unit Step Input

$$e_k = r_k - (1-a)r_{k-1} - ar_{k-2} + (1-a)e_{k-1} + ae_{k-2} - q_0(1+a)e_{k-6} - q_1(1+a)e_{k-7} - q_2(1+a)e_{k-8}$$

EQN 21

C.2 Second Order System

The `CostSecond` function is a pre-existing function that was developed as part of a previous assignment. Whereby the open-loop response $c(t)$ to a unit ramp input can be calculated from the transfer function $G_p(s)$, such that:

Given,

$$G_p(s) = \frac{C(s)}{M(s)}$$

EQN 22

And,

$$G_p(s) = \frac{(s+3)}{(s^2 + 2.s + 109)}$$

EQN 23

Then,

$$\begin{aligned} C(s) &= G(s).M(s) \\ &= \left\{\frac{(s+3)}{s^2 + 2.s + 109}\right\} \times \left\{\frac{1}{s}\right\} \end{aligned}$$

EQN 24

$$c(t) = \text{Inverse Laplace of } \left[\left\{\frac{s+3}{s^2 + 2.s + 109}\right\} \times \left\{\frac{1}{s}\right\}\right]$$

EQN 25

The `CostSecond` function uses ITAE Criterion.

Using the sampling interval of 0.05s, $G_{hp}(z)$ numerically simplifies to:

$$G_{hp}(z) = \left[\frac{A + B.z^{-1} + C.z^{-2}}{1 + D.z^{-1} + E.z^{-2}}\right]$$

EQN 26

Where,

$$\begin{aligned} A &= 0.0275 \\ B &= 0.0171 \\ C &= -0.0383 \\ D &= -1.5107 \\ E &= 0.7408 \end{aligned}$$

The PID controller transfer function can be written as:

$$G_c(z) = \left[\frac{q_0 + q_1.z^{-1} + q_2.z^{-2}}{1 - z^{-1}} \right]$$

EQN 27

This can be written in the form of a difference equation:

$$m_k = q_0.e_k + q_1 e_{k-1} + q_2.e_{k-2} + m_{k-1}$$

m_k is calculated by solving the simultaneous equations:

$$\begin{aligned} E(z) &= R(z) - C(z) \\ C(z) &= G_{hp}(z).M(z) \\ M(z) &= G_c(z).E(z) \end{aligned}$$

EQN 28

These simultaneous equations were converted to difference equations. Then the simultaneous equations were solved by setting up the matrix equation $AX = B$ where:

$$A = \begin{bmatrix} 1 & 1 & 0 \\ -q_0 & 0 & 1 \\ 0 & 1 & -a \end{bmatrix}$$

$$X = [e(k) \quad c(k) \quad m(k)]$$

$$B = \begin{bmatrix} b_1 \\ b_2 \\ b \end{bmatrix}$$

$$\begin{aligned} b_1 &= r(k) \\ b_2 &= q_1.E(k-1) + q_2.E(k-2) + M(k-1) \\ b_3 &= b.M(k-1) + c.M(k-2) - d.C(k-1) - e.C(k-2) \end{aligned}$$

EQN 29

The built-in MATLAB® function `fminsearch` indicated that the global minimum was obtained by the parameter values:

$$\begin{aligned} q_0 &= 878.162019162082 \\ q_1 &= 1127.82834366859 \\ q_3 &= 464.06631121099 \end{aligned}$$

EQN 30

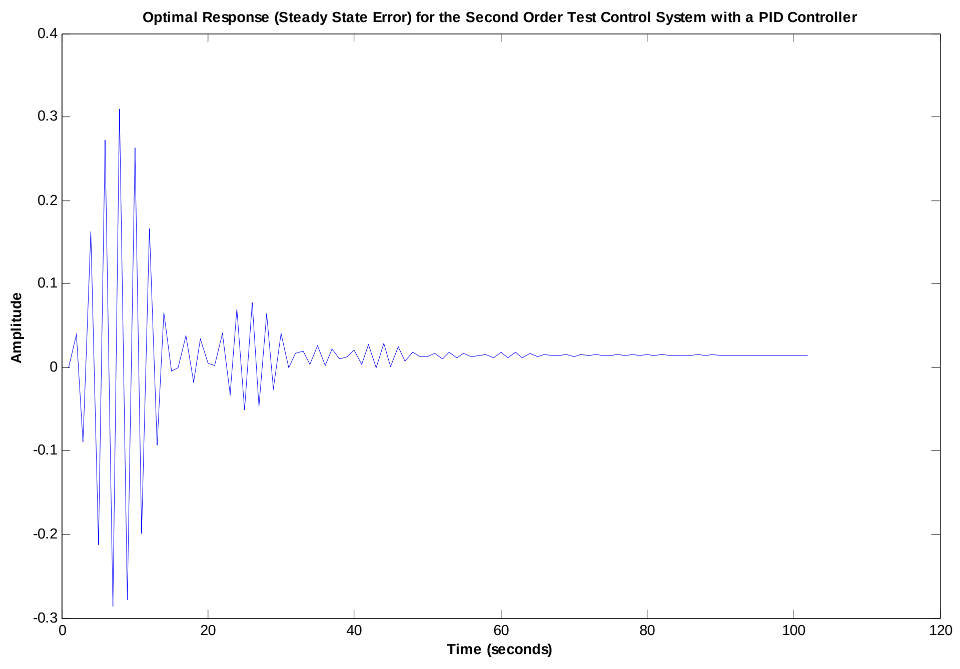


Figure C -3 Optimum Response (Steady State Error) for the Second Order Test Control System with a PID Controller and Unit Ramp Input

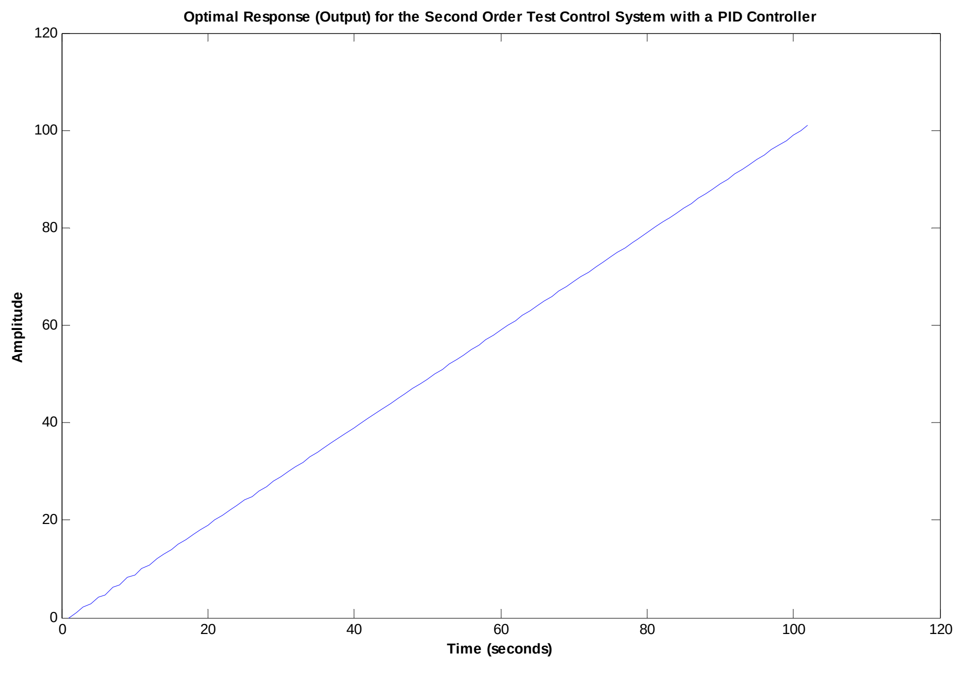


Figure C -4 Optimum Response (Output) for the Second Order Test Control System with a PID Controller and Unit Step Input

C.3 First Order System – Random Delayed Input

The `CostFirstRandomDelay` function is a modification of the CostFirst function with the addition of a random delay to input. The random delay is defined main the main routine using a mean and standard deviation.

Table C -1 Random Delay Calculation

```
Random_Delay=round(AVERAGE+STDDEV.*randn);
```

The difference equations for both the error (refer EQN 16) and output (refer EQN 18) have been modified to support a random delay, resulting in new difference equations:

$$e_k = r_k - (1-a)r_{k-1} - ar_{k-2} + (1-a)e_{k-1} + ae_{k-2} - q_0(1+a)e_{k-x} - q_1(1+a)e_{k-1-x} - q_2(1+a)e_{k-2-x}$$
$$\text{where } a = -e^{-0.5}\text{; and } x \text{ is random time delay to system}$$

EQN 31

$$c_k = (1-a)c_{k-1} + ac_{k-2} + q_0(1+a)e_{k-x} + q_1(1+a)e_{k-1-x} + q_2(1+a)e_{k-2-x}$$
$$\text{where } a = -e^{-0.5}\text{; and } x \text{ is random time delay to system}$$

EQN 32