Evolution of Robotic Controllers
Using Genetic Algorithms

Mark Beckerleg

A thesis submitted to AUT University
in fulfilment of the requirements for the degree of
Doctor of Philosophy (PhD)

March 13, 2012

School of Engineering
Primary Supervisor: John Collins

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been accepted for the award of any other degree or diploma of a university or other institution of higher learning.

Signed……...

Dedication

I dedicate this thesis to

- My mother Loloma Florence Beckerleg and my father Brazillia Sturges Beckerleg both of whom would be very proud of this achievement.
- My family Robyn and Peter Beckerleg.
- My loving partner Sarah Moore.

Acknowledgement

I would like to acknowledge my supervisor John Collins for his support and encouragement in this thesis; one of the great minds in the school of engineering, my partner Sarah Moore for her endless patience and superb writing skills and my sister Robyn Beckerleg.

Abstract

This research investigates evolutionary robotics which uses evolutionary computation to generate robotic controllers. The majority of research in this field has been primarily focused on the use of software genetic algorithms to evolve robotic controllers based on artificial neural networks and fuzzy logic. Investigation into other forms of evolvable robotic controllers however is less studied, thus the focus of this research was to investigate and develop new methods of evolving controllers for evolutionary robotics. This led to the creation of three novel concepts within this field including the evolution of lookup tables for robotic control, the implementation of the robotic simulation in hardware for fitness evaluation of individuals, and advances in virtual Field Programmable Gate Arrays (FPGAs) for robotic control.

The innovative utilization of a lookup table for a robotic controller used multi-dimensional lookup tables that linked the state of the robot obtained from input sensors to the required output for the robots actuators in order for the robot to function correctly. A population of these tables (chromosomes) were evolved using genetic algorithms. Two multi-dimensional lookup table robotic controllers were successfully evolved using standard genetic algorithms.

The novel approach of implementing the robotic simulation in hardware rather than software was performed. The time required for a genetic algorithm to evolve a successful robotic controller is largely dependent on the fitness evaluation of an individual. If the robotic simulation could be performed in hardware then there will be a significant increase in performance. It was shown that hardware robotic simulations could be constructed with an improvement in evolution completion time of over two orders of magnitude greater than that of a software simulation.

The use of robotic controllers in the form of two virtual FPGAs were evaluated using two Cartesian based architectures, a fixed layer and a reducing layer virtual FPGA. The configuration bit stream which describes the circuits within the virtual FPGA was evolved by a genetic algorithm implemented in hardware. The input sensors of the robot, indicating its current state were connected to the inputs of the virtual FPGA, while the output was connected to the robot actuators. It was found that both architectures could be evolved to produce robotic controllers.

Table of Contents

Table of Figures

Tables

Chapter 1

Chapter 1: Introduction

The focus of this research was to investigate and develop new methods of evolving controllers for evolutionary robotics. Areas that were explored were the use of evolvable lookup tables and virtual FPGAs for robotic control, and the development of high speed hardware based simulations for fitness evaluations. As a result, two novel evolutionary capable robotic controllers, one based in software and the other in hardware were developed. A further outcome was the creation of a hardware robotic simulation that outperformed a similar software robotic simulation by two orders of magnitude.

Within the field of study known as robotics, the term robot is defined in a multitude of ways. For the purposes of this thesis a 'robot' is deemed to be an electro-mechanical machine which can be programmed to interact with physical objects to perform specific actions either semi or fully autonomously. In contrast, a machine such as a mechanical pump has little processing control and achieves its task through a simple mechanical process. The more an electro-mechanical machine seems to have a purpose of its own, the more likely it is to be called a robot as it conveys a sense of intent. As such, a clock with its preset motions and inability to adapt to changes in its environment is not considered to be a robot, whereas an autonomous car which can sense and react to its environment is considered to be a robot. Gregory Dudek [1], the director of the Centre for Intelligent Machines at McGill University in Montreal, sets three criteria for robots: a) robots need to make measurements of the environment around them, b) they will follow a program which makes decisions, and c) they will take actions depending on the environment and the robots programmed decisions.

Robots are utilised for specific purposes within a diverse area of applications such as industry (car production, packaging, electronics, transportation of goods within warehouses and container ports), agriculture (fruit harvesting, sowing and fertilizing), domestic use (vacuuming, lawn mowing and floor polishing), military (unmanned

combat air vehicles and unmanned aircraft vehicles), human safety (bomb disposal and nuclear handling), and within the healthcare sector (pharmaceutical production, mobility scooters and other such support for disabled and elderly people).

Evolutionary robotics is a sub-field of robotics which uses evolutionary computation to generate the controller for the robot. It mimics biological evolution to evolve a robotic controller, thus enabling the robot to interact with its environment without the direct coding of the robots tasks by human programming instructions. In addition, this enables fault tolerant controllers which can adapt to environments that change beyond the programmer's expectations. Evolutionary robotics applies evolutionary computation to robotic controllers such as artificial neural networks, fuzzy logic, proportional-integral-derivative (PID), and evolvable hardware. It can be applied to most robotic systems; however the majority of research is focused in the area of autonomous robots that work without human intervention in unstructured environments. There are a range of areas within evolutionary robotics, such as control of manipulators, path planning, obstacle avoidance, behaviour based control and morphogenesis.

Evolutionary computation is an optimization process that autonomously searches through a set of possible solutions to a problem, to find a solution that will adequately solve the problem. It applies the complex concepts embedded in the theory of biological evolution developed by Charles Darwin wherein a population of organisms, through the processes of variation, selection and heredity, evolve and adapt to their environment. In the case of evolutionary computation, the population is comprised of solutions to a problem which will evolve. Each solution is evaluated and given a fitness rating, and the solutions with a higher fitness are retained and used to create new solutions. These solutions are often referred to as individuals or chromosomes, while a group of solutions is called a population. Chromosomes can be in many forms depending on the problem to be solved. Evolutionary computation is used in a diverse range of areas and not limited to evolutionary robotics. There are several forms of evolutionary computation, one of which is the genetic algorithm, which is discussed in detail in chapter two.

The genetic algorithm is a repetitive process with three parts including a) reproduction, where the genetic operators crossover and mutation are used to generate new individuals from the surviving population of individuals, b) fitness evaluation, which determines how well each individual within the population performs, and c) selection, which is the

process that determines which individuals within the population (based on their fitness) will survive to the next generation.

1.1 Research Objectives

The objective of this thesis was to explore new methods that could be used in evolutionary robotics. This led to the investigation into the improvement of two major components of evolutionary robotics, these being the robotic controllers themselves and the genetic algorithms used to evolve them. This research resulted in three significant developments. These included the creation and evaluation of an evolutionary capable robotic controller based on a lookup table, an evolutionary capable robotic controller based on virtual FPGAs and a hardware based high speed simulation used within a genetic algorithm.

As a result of the above investigations, the following list of research questions arose:

- Can a lookup table be evolved to function as a robotic controller?
- Can a virtual FPGA be evolved to function as a robotic controller?
- Can the genetic algorithm's simulation be implemented in hardware?

These questions and the systems used to evaluate them are described in the following section.

1.1.1 Can a lookup table be evolved to function as a robotic controller?

The concept of using a lookup table as a robotic controller and configuring their parameters using a genetic algorithm was evaluated. Software genetic algorithms were developed to evolve two robotic controllers; the first to evolve a controller for a robotic mobile inverted pendulum, and the second to evolve a controller for a robotic ball-balancing beam. In both cases, the heart of the controller was a lookup table. The axes of the lookup table were linked to the state of each robot. The parameters within each lookup table were the required motor speed and direction needed to control the robotic motion.

In the case of the mobile inverted pendulum, the lookup table was a two dimensional array with one axis relating to the pendulum angle, and the other axis relating to the pendulum's angular velocity. The parameters in the lookup table provided the motor speed and direction required to keep the pendulum in balance. In the case of the ball-balancing beam, a three dimensional lookup table was evolved with the first axis relating to the ball position, the second axis relating to the ball speed, and the third axis relating to the beam position. The parameter at each lookup table location described the motor speed and direction required to keep the ball balanced on the beam. The genetic algorithm, robotic simulation, and lookup tables were contained within a computer as shown in Figure 1-1.

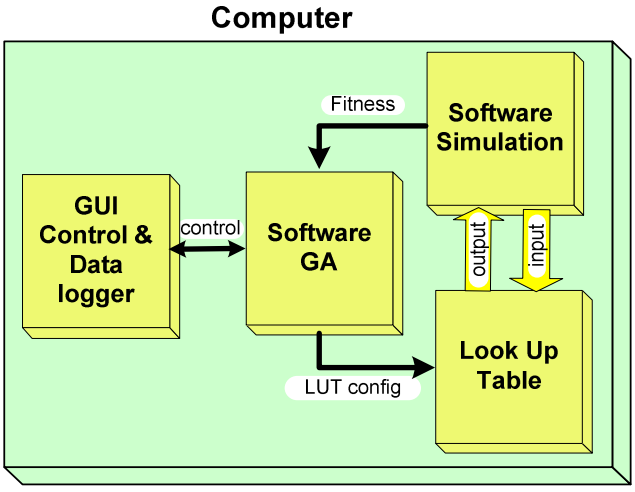


Figure 1-1. Software genetic algorithm using a lookup table and software simulation based in a computer.

1.1.2 Can a virtual FPGA be evolved to function as a robotic controller?

To determine if a robotic controller implemented as a virtual FPGA could be evolved, both a hardware genetic algorithm and a virtual FPGA were designed and constructed. The virtual FPGA was used as a robotic controller whose task was to balance a ball on a beam. The virtual FPGA was evolved by performing a genetic algorithm on its configuration bit stream. The virtual FPGA was a hardware circuit that was specifically designed to suit the evolutionary process and could be created within a FPGA. A hardware genetic algorithm was created which was capable of evolving the configuration bit stream of the virtual FPGA. The virtual FPGA, hardware GA and NIOS processor were contained inside a FPGA, with a communication link to a computer as shown in Figure 1-2. The computer was used for data logging and graphical display of the ball and beam. The virtual FPGA was custom designed to suit evolution and replicated an 'ideal' FPGA. Due to limited FPGA resources, the hardware genetic algorithm used the mutation genetic operator without the crossover operator. The robotic simulation was executed on a NIOS processor located inside the FPGA. The virtual FPGA was connected to the robotic simulation output states via a 32 bit bus, sending the desired motor speed to the simulated beam motor. As well as executing the robotic simulation, the NIOS processor controlled the actions of the hardware genetic algorithm. The robotic platform that was used to evaluate the virtual FPGA and hardware genetic algorithm was the ball-balancing beam.

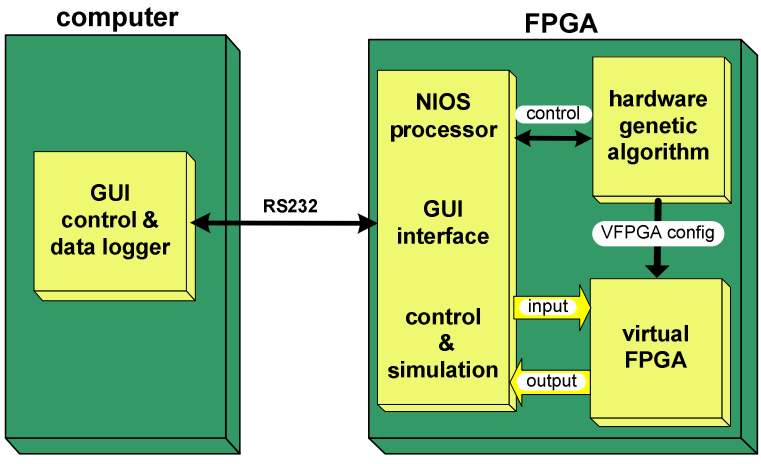


Figure 1-2. System interconnections for the virtual FPGA and a hardware genetic algorithm located inside a FPGA.

1.1.3 Can a simulation used for a genetic algorithm be implemented in hardware and benefit the evolutionary process?

The process that consumes the most time within a genetic algorithm is fitness evaluation of individuals within a population. Any increase in speed in this process would significantly reduce the time taken for a genetic algorithm to reach a suitable solution. The fitness evaluation is normally performed using a robotic simulation executed on a computer with the simulation's mathematical calculations performed sequentially. Alternatively if the simulation could be implemented in hardware with its associated parallelism rather than executed in software, the mathematical calculations could be performed in parallel giving an associated improvement in performance with a large improvement in the speed of the fitness evaluation. To investigate this question, a comparison between an integer simulation running on the NIOS processor and a hardware simulation was performed as shown in Figure 1-3.

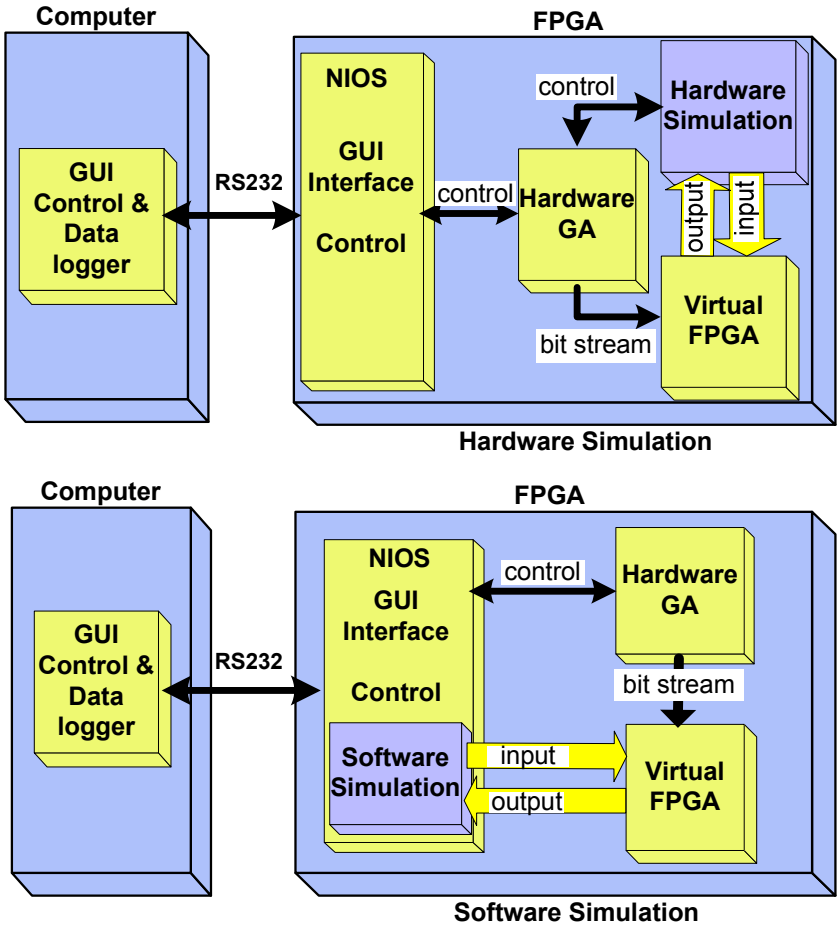


Figure 1-3. System interconnections for the hardware simulation and a software simulation running on a NIOS processor both located in an FPGA.

1.2 Publications

A range of material generated from this thesis has been published in peer reviewed international conference, book chapters and journal articles:

1.2.1 Conference Papers

- M. Beckerleg, J Collins, "A GA based Controller for a Mobile Inverted Pendulum," in *ICARA The Third International Conference on Autonomous Robots and Agents* Palmerston Nth, New Zealand, 2006. (Contribution MB 80% JC 20%)
- M. Beckerleg, J Collins, “Evolving Electronic Circuits for Robotic Control” in *Mechatronics and Machine Vision in Practice, 2008. M2VIP 2008. 15th International Conference on*, 2008. (Contribution MB 80% JC 20%)
- J. Currie, M. Beckerleg, and J. Collins, "Software Evolution Of A Hexapod Robot Walking Gait," in *Mechatronics and Machine Vision in Practice, 2008. M2VIP 2008. 15th International Conference*, 2008. pp. 305-310. (Contribution JC 50% MB 25% JC 25%)
- M. Beckerleg, J. Collins, “Using a Hardware Simulation within a Genetic Algorithm to evolve Robotic Controllers” in International Conference on Intelligent Automation and Robotics (ICIAR'11) 2011. (Contribution MB 90% JC 10%) [Recommended best paper award and journal publication].
- M. Beckerleg, J. Collins, “Evolving a Three Dimensional Look Up Table Controller for a Curved Ball and Beam System” in International Conference on Intelligent Automation and Robotics (ICIAR'11) 2011. (Contribution MB 90% JC 10%) [Recommended best paper award and journal publication].

1.2.2 Book Chapters

- M. Beckerleg, J. Collins, "An Analysis of the Chromosome Generated by a Genetic Algorithm Used to Create a Controller for a Mobile Inverted Pendulum," *Studies in Computational Intelligence,* vol. 76, 2007. (Contribution MB 80% JC 20%)

1.2.3 Journals

- J. Currie, M. Beckerleg, and J. Collins, "Software Evolution Of A Hexapod Robot Walking Gait," in *International Journal of Intelligent Systems Technology*, 2010, pp. 382-394. (Contribution JC 50% MB 25% JC 25%)

1.3 Thesis Structure

This thesis is organised as follows:

CHAPTER 1**:** This chapter explains the motivation for the thesis, the questions that the thesis explores, and an overview of the systems that were developed to test the hypotheses.

CHAPTER 2: An introduction to the four areas of evolutionary computation is provided along with an explanation of the concepts of a genetic algorithm. Genetic reproduction, selection and fitness evaluation are investigated and the latest research in these areas is discussed, with a particular emphasis on robotic controllers.

CHAPTER 3**:** This chapter explains the concepts of hardware evolution, describing its associated problems and solutions. The use of a virtual FPGA for hardware evolution is discussed with a focus on their application in robotic controllers.

CHAPTER 4**:** An overview of artificial neural networks and fuzzy logic is provided, detailing how they can be adapted for evolution. The evolutionary process is explained in terms of evolution of the weights in an artificial neural network, and evolution of the rules and classes in fuzzy logic. Examples of robotic controllers are described in both cases.

CHAPTER 5**:** This chapter reviews current research that utilised the mobile inverted pendulum and the ball-balancing beam, with particular focus on the mathematical modelling of these devices and the systems used to control them.

CHAPTER 6: This chapter explains common systems that were developed for this thesis. These include mathematical models of the mobile inverted pendulum and ball-balancing beam, the implementation of the model in simulation, the graphical user interfaces for data recording and control, and the data communication protocol between computer and FPGA.

CHAPTER 7: A description is provided of how two robotic controllers based on lookup tables were created and evaluated for their evolutionary capabilities. The associated problems and solutions are described and the results presented.

CHAPTER 8**:** This chapter explains how a virtual FPGA was designed and evolved for the controller of the ball-balancing beam. A description is provided about the hardware

genetic algorithm and the virtual FPGA that were used in the evolutionary process. Experimental results are presented.

Chapter 9**:** A description is provided of how a simulation's performance could be increased by moving its mathematical equations from software to hardware. Identical simulations were created in both software and hardware and their performance evaluated.

CHAPTER 10: Conclusions are drawn and future research potentials identified.

Chapter 2

Chapter 2: A Review of Software Genetic Algorithms and their use in Evolutionary Robotics

Evolutionary computation is used to solve combinatorial optimization problems. Based on Charles Darwin's biological theory of evolution, it utilizes an iterative process wherein populations of individuals evolve to best suit their environment. Evolutionary computation is employed to autonomously search through a sequence of possible solutions to a problem to eventually find a solution that will adequately solve that problem. It will not however necessarily find the optimum solution.

To briefly summarise the tenets of biological evolution: there exist populations which are groups of individuals of a species. Each individual possesses a genome which consists of chromosomes which in turn contain a large number of genes. These genes encode for, or control, inherited traits. The complete genome for an individual is known as its genotype; the complete set of observable traits its phenotype. Individuals mate and reproduce offspring thereby transmitting traits from one generation to the next. During the process of reproduction specific mechanisms such as natural selection, genetic mutation, genetic recombination or gene flow enable genetic variation or change. Any change that enhances the individual's traits becomes and remains more common in successive generations of a population. More offspring are produced than the environment can support and these offspring vary in their ability to survive and reproduce. As a result, competition for survival and reproduction ensues. Individuals with favourable traits which are best adapted to their environment possess, in evolutionary terms, greater 'fitness'; as such, they are able to survive, reproduce and transmit their genetic characteristic in increasing numbers to succeeding generations. Over time, these 'fitness' traits become dominant within the population.

This biological process of evolution is replicated in evolutionary computation where the chromosome may describe engineering or commercial parameters such as an aircraft's landing schedule. Changing the chromosome will modify the landing schedule, and by

applying an evolutionary process to the chromosome, the landing schedule will evolve. Evolutionary computation is used in a wide range of commercial applications such as scheduling, bio-computing, economics, financial market analysis [2], telecommunications, imaging, integrated circuit design and drug design. Evolutionary computation consists of four broad areas: genetic algorithms, genetic programming, evolutionary strategies and evolutionary programming.

2.1 Evolutionary Computation Techniques

2.1.1 Genetic Algorithms

The interest in genetic algorithms began in the 1960's due to the work of John Holland [3, 4]. Holland's principal aim was to increase the understanding of the natural evolutionary process, and to use these techniques in the design of man made systems.

Genetic algorithms are used to find solutions to a problem using natural selection as a search engine, for example, the problem could be to create an aircraft landing schedule at an airport. They act on a population of individuals or chromosomes. Within this population, these chromosomes are potential candidate solutions to the problem needing to be solved. Chromosomes are comprised of various forms such as bits, numbers or parameter sequences, depending on the problem.

The genetic algorithm is iterative and is comprised of three main processes: reproduction, fitness evaluation and selection.

- Reproduction is the generation of offspring from the surviving population of individuals or chromosomes. Reproduction uses two genetic operators: crossover, where chromosomes are exchanged between parents, and mutation, where parts of the parents' chromosomes are randomly altered.
- Fitness evaluation determines how well each individual or chromosome in the population performs as a potential solution to the problem.
- Selection determines which individuals or chromosomes within the population will survive to the next generation based on their fitness.

The steps in a genetic algorithm are shown in Figure 2-1. Initially, a random population of chromosomes (candidate solutions) is generated. The population of chromosomes is then used to produce offspring by combining the chromosomes of the parents using crossover and mutation (reproduction). Each new chromosome is then evaluated to

determine how well it solves the assigned problem, and subsequently a fitness rating (fitness evaluation) given. The chromosomes with the best fitness are kept (selection) and used to create new offspring. The processes of reproduction, fitness evaluation and selection are repeated until the required fitness is reached or a set number of generations have been completed.

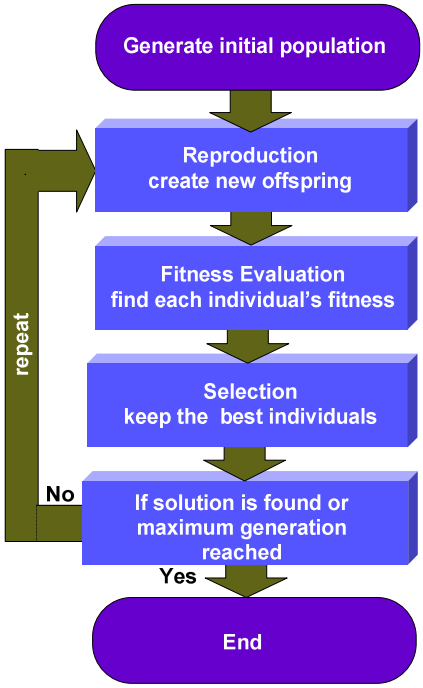


Figure 2-1. Flow chart describing the genetic algorithm process.

Two examples of using a genetic algorithm to create a solution to a problem are provided. The first example uses the classic Travelling Salesman non-deterministic polynomial as the problem to be solved. The second example shows how a genetic algorithm can be used to generate a nurse work roster.

Travelling Salesman

The travelling salesman wishes to visit ten cities in New Zealand as listed in Table 2-1. The salesman wants to visit each city only once and then return home. The task is to find the shortest route he can take for his journey. The candidate solutions (chromosomes) will be a list of numbers with each number representing a city. The sequence that the numbers appear in the chromosome is the order that the cities will be visited. One possible chromosome would be [10, 2, 8, 6, 5, 1, 4, 9, 7, 3]. The fitness level would be judged on which chromosome described the shortest distance that the salesman would be required to travel.

Auckland	1	Rotorua	6
Hamilton	2	Tauranga	7
Wellington	3	Whangarei	8
Christchurch	4	New Plymouth	9
Dunedin	5	Invercargill	10

Table 2-1. List of cities and their associated numbers to be visited by the travelling salesman.

This problem is referred to as a non-deterministic polynomial as it is difficult to know if the best solution is found since the number of possible solutions is so large. The number of possible solutions to this problem is the factorial of the number of cities. This number will rapidly increase as the number of cities visited by the salesman also increases. The number of possible solutions or chromosome permutations is referred to as the search space. In this example, the search space is factorial 10 which gives approximately 3.6 million solutions. If 50 cities were visited, the search space would expand to $3x10^{64}$.

Due to the reproduction process of crossover and mutation, it is possible that the new offspring will be damaged as the chromosome may contain the same city twice, or it might not include all the cities. Thus, after each reproduction cycle, the new chromosome needs to be checked and repaired if found to be damaged.

Nurse Roster

The second example uses a genetic algorithm to generate a nursing roster for nurses working twelve hour shifts at a hospital. The two variables involved are the nurses themselves and the times that they are required to work. The chromosome would be a sequence of letters where the letter represents the nurse, while the order within the sequence indicates the time they are required to work. In this example, there are five nurses, Sarah (S), Mark (M), Bronwyn (B) Ricky (R) and Terry (T). The times that the nurses are required to work are shown in Table 2-2. A possible chromosome would appear as [B, R, S, M, T, B, R, S, M, T, B, R, S, M].

Position in chromosome	0	1	2	3	.....	12	13
Time	Monday 7am-7pm	Monday 7pm-7am	Tuesday 7am-7pm	Tuesday 7pm-7am	.....	Sunday 7am-7pm	Sunday 7pm-7am

Table 2-2. An example of a chromosome for the nurse roster.

The fitness evaluation would be determined by how well the chromosome met the roster criteria. This criteria would depend on the rostering requirements, for example, a nurse could not work more than one 12 hour shift in a day, a nurse could not work more than

three shifts per week, and it would be preferred that the nurses work three day shifts or three night shifts in a row. A more advanced fitness evaluation would allow nurses to request days or weeks off for scheduled holidays. In this case, although a chromosome might be poor, no newly generated offspring would be illegal, and thus chromosome repair would never be required.

2.1.2 Genetic Programming

Genetic programming was initially developed by John Koza [5]. This process is similar to a genetic algorithm except the chromosome is a computer program rather than a possible solution as used in a genetic algorithm. Thus genetic programming is used to evolve a computer program. The program is represented as a tree structure with the branches of the tree representing the functions of the program, while the leaves on the branches represent the variables and constants. It is this tree structure that is altered by the evolutionary process. The fitness is evaluated by running the program to see how well it performs.

Figure 2-2 shows how an original program ((x+1)(y+1)) + ((x-2) + (y-2)) can be represented in a tree structure. The variables and constants are listed on the bottom of the tree with the arithmetic operators in the branches. In this example x and one are added together, y and one are added together then these two branches are multiplied to give (x+1)(y+1). Similarly, 2 is subtracted from x and 2 is subtracted from y. These two branches are added to give (x-2) + (y-2). It is this tree structure that is evolved producing a new program for each generation.

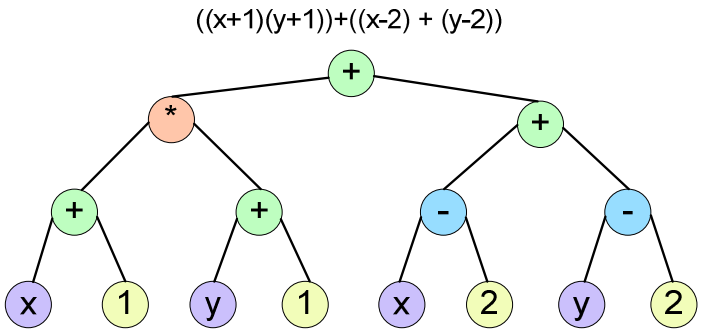


Figure 2-2. Diagrammatical representation of a tree structure in genetic programming.

2.1.3 Evolutionary Strategies

Evolutionary strategy was developed by Rechenberg [6, 7] and Schwefel [8] at the Technical Institute of Berlin. It is designed to solve technical optimization problems

(initially wind tunnel experiments) where the individuals represent the phenotype (characteristics) of the object to be optimised. These individuals are real numbers and are evolved using the same operators as a genetic algorithm (crossover and selection). The amount of mutation is however determined by a Gaussian distributed random value. In general, evolutionary strategies use an adaptive mutation rate, where the mutation rate was part of the chromosome and has a possibility of changing after each generation. If the offspring has a better fitness it will replace the parents.

2.1.4 Evolutionary Programming

Evolutionary programming was developed by Fogel et al. in the early 1960's [9, 10] primarily as a way of achieving artificial intelligence through evolution, but has since been used for optimization problems. The term evolutionary programming came from initial experiments where a population of finite state machines were evolved to describe the behaviour of a software program. Later, these individuals were extended to real problem domains such as real-value vectors, ordered lists, trees or finite state machines. The evolutionary process uses only the mutation operator as the crossover operator is not normally used.

2.2 Fitness Landscape

The fitness landscape is a graphical representation showing the fitness of all possible combinations of chromosomes. It is a two dimensional graph with the fitness represented on the Y-axis and all possible chromosomes represented on the X-axis as shown in Figure 2-3. The size of the fitness landscape is proportional to the size of the chromosome, with a larger chromosome having a larger fitness landscape.

Possible solutions or successful chromosomes are known as maxima on the graph. Most problems will have multiple maxima. The fitness of chromosomes between maxima is non linear, thus the fitness landscape becomes a succession of peaks and troughs. As the chromosomes evolve, they will move up the fitness landscape towards a maximum. However, they will often move to peaks known as local maxima, where an improvement in fitness is found, but not a final solution. The evolution may come to a halt at local maxima, unless the selection and reproduction process can move the chromosome into the adjacent trough and up to the next peak.

In Figure 2-3, points A and B are local maxima with C being the optimum solution. The arrows indicate the evolutionary direction as the population moves from a very low fitness value to the top of a peak. In order for a chromosome to move beyond the local maxima at A, a large diversity in the population is required which is dependent on the type of selection process used, and a mutation rate high enough to allow the chromosomes to move down the trough and up the adjacent peak.

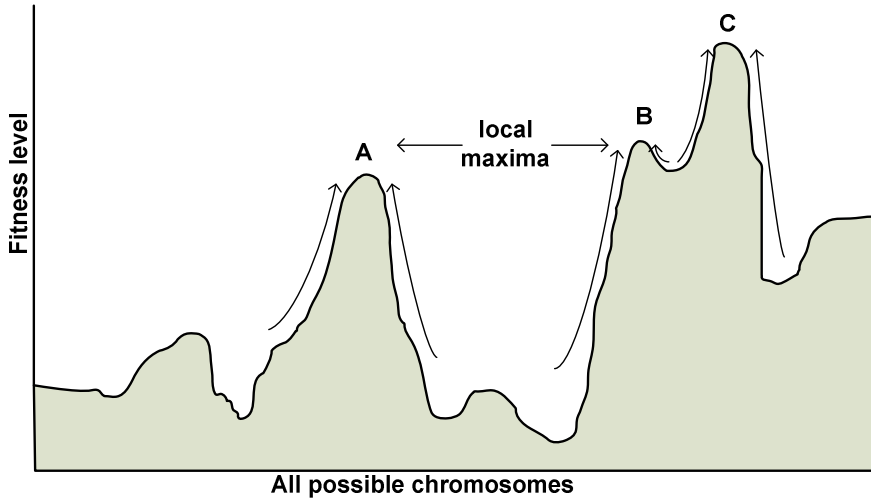


Figure 2-3. Graphical representation of the fitness landscape.

Figure 2-4 shows how the selection process retains the chromosomes with the higher fitness, thus the fitness moves up the fitness landscape. The function of the mutation process is to prevent the chromosomes from becoming trapped at local maxima. The level of mutation required to move the chromosome beyond local maxima can be calculated from the hamming distance. The hamming distance describes the minimum number of changes required within the chromosome to move from the peak of the local maxima to the adjacent trough.

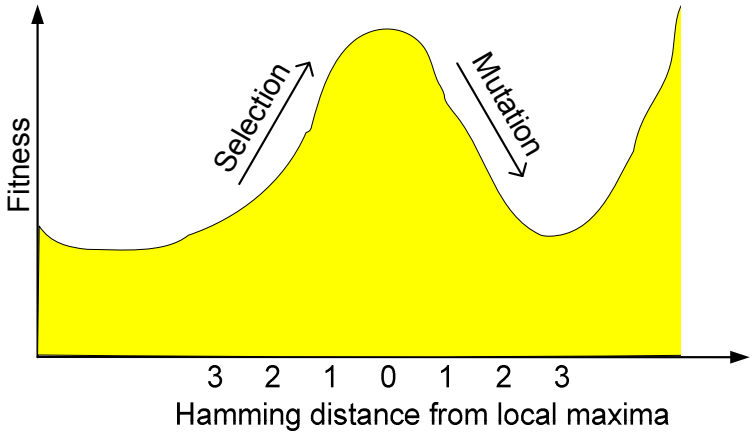


Figure 2-4. Fitness landscape showing the hamming distance from the local maxima.

2.3 Genetic Reproduction

The genetic algorithm has three processes: reproduction, fitness evaluation, and selection. The reproduction techniques crossover and mutation are used to alter the chromosome as demonstrated in this section. The following examples relate to the travelling salesman problem where the chromosome is configured for ten cities. The initial parents are shown in Figure 2-5.

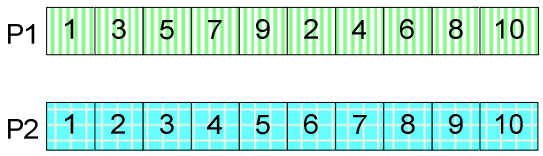


Figure 2-5. The initial chromosome for a travelling salesman ten city journey.

2.3.1 Crossover

Crossover is a method that is used to split and "recombine" the chromosomes from two or more parents into one offspring. The standard crossover algorithms are described below.

Single-point crossover: A point at random in one parent's chromosome is chosen and the parameters in the chromosome after this point are swapped with the parameters of the other parent's chromosome. This process creates an offspring with a combination of both parents' chromosomes as shown in Figure 2-6. Note in this case the resultant chromosome (child) is invalid as the same city (9) occurs twice within it. This chromosome will need to be repaired (repaired child), so that it becomes valid. The disadvantage to single point crossover is that both the beginning and end of one parent's chromosome may contain good properties but cannot both be passed to one offspring because only the first or last part of any parent will be transferred to the offspring.

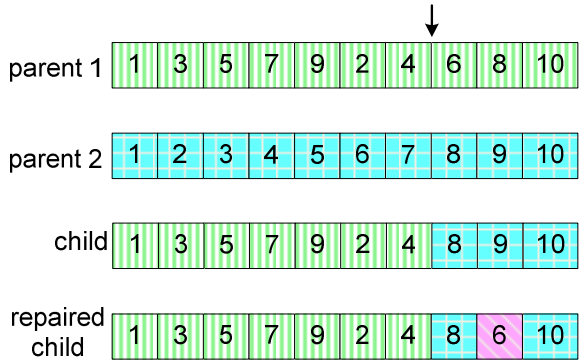


Figure 2-6. An example of a chromosome after single point crossover.

Two-point crossover: Two points in the chromosome are randomly chosen and the chromosomes inside these two points are swapped between the two parents' chromosomes, as shown in Figure 2-7. This is considered to be a better technique than single-point crossover as it overcomes the problem that the single-point crossover has in not being able to transfer the first and last part of its chromosome.

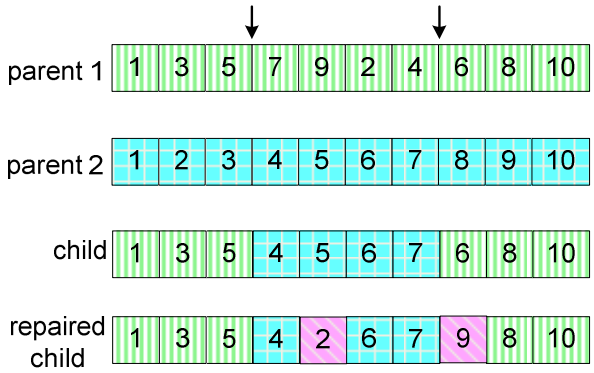


Figure 2-7. An example of a chromosome after two point crossover.

Multiple-point crossover: Multiple points are chosen at random and the chromosomes between these points are exchanged as shown in Figure 2-8. The advantage of this technique is that the search space can be more thoroughly explored; however, sequences or building blocks within the chromosome may be destroyed as good sequences within the chromosome may be split.

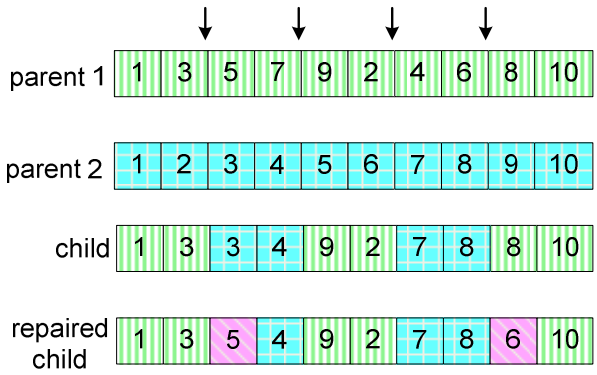


Figure 2-8. An example of a chromosome after multiple point crossover.

Uniform crossover: Each parameter within the parents' chromosome will have a 50% probability of being passed to the offspring as shown in Figure 2-9. Although it can be disruptive, to the chromosome uniform crossover has a strong bias towards exploration of the search space. This method is particularly useful if the population size is small as it produces a high variation between parents and offspring, due to the chromosome being altered across its complete range.

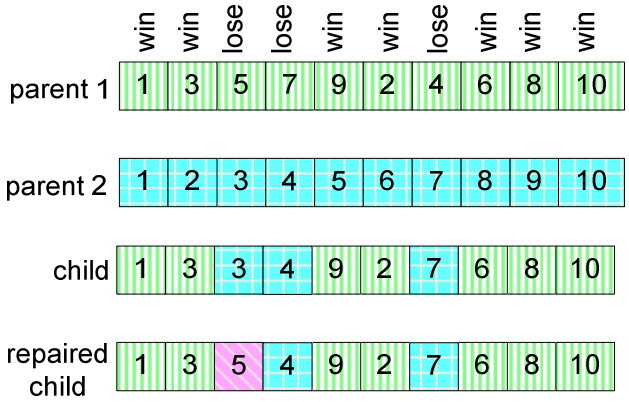


Figure 2-9. An example of a chromosome after crossover uniform.

There are many other methods of crossover such as a) arithmetic, where an arithmetic crossover operator is applied to the two parents to form an offspring, b) heuristic multiple parents using diagonal crossover and scanning crossover [11], and c) multiple parents using polynomial and lognormal distribution[12].

2.3.2 Mutation

Mutation is a reproduction method that operates on a single chromosome. It is a random process and has a low probability of occurring which is typically between 0.1 to 2 percent. Its purpose is to increase the diversity of the population. There are many mutation techniques, some of which are described in this section.

Insertion Mutation: A random point within the parent's chromosome is selected and randomly altered. A variation of insertion mutation is creep mutation, where the parameter that is to be mutated will be replaced with a value that is within a percentage range of the original parameter rather than with a random value.

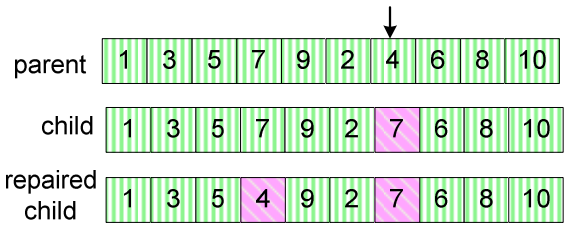


Figure 2-10. An example of a chromosome after insertion mutation.

Inversion Mutation: Two random points are selected in the parent's chromosome and the parameters between the selected points are inverted. This mimics a naturally occurring biological mutation.

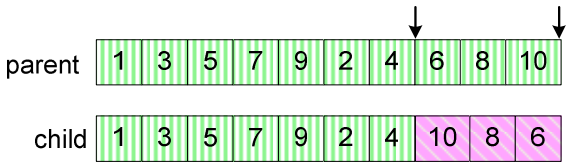


Figure 2-11. An example of a chromosome after inversion mutation.

Exchange Mutation: Two random points are selected in the parent chromosome and the parameters at that point are swapped.

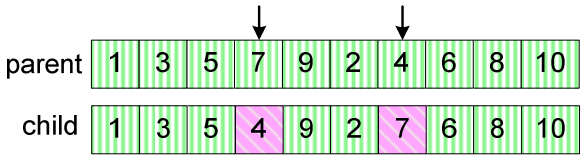


Figure 2-12. An example of a chromosome after exchange mutation.

Displacement Mutation: Two random points are selected, and the parameters between these points are moved to a randomly selected part of the chromosome.

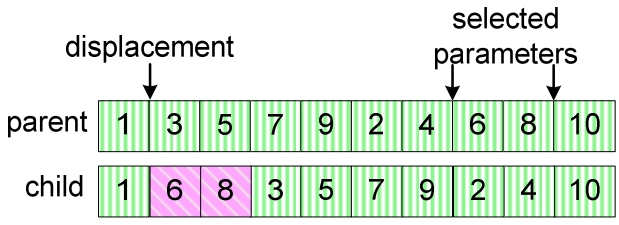


Figure 2-13. An example of a chromosome after displacement mutation.

In general it has been found that the mutation rate should be high to begin with but then should be reduced as the evolution approaches a high fitness. Muhlenbein [13] studied bit-string chromosomes and found that as more bits within the chromosome string were correct, the probability of a mutation producing a better offspring decreased. Conversely, when more bits within the chromosome string were correct, the probability of crossover producing a better offspring increased. It was found that if a constant mutation rate was used, then the optimum mutation rate was 1/L where L was the bit string length. The use of the mutation operator has normally been performed in conjunction with the crossover operator. However, there are algorithms which use the mutation operator only.

2.4 Genetic Selection

Genetic selection is the method used to select which offspring will be kept to become parents for the next generation. Its purpose is to move the population to a higher fitness level. Genetic selection uses the fitness value of each offspring to determine if it will be retained. Each offspring is evaluated and given a fitness, which is used by the selection process. The selection pressure is an important parameter in genetic selection. The selection operator has a high selective pressure if it severely reduces the difference between individuals or a low selective pressure if it allows many different individuals to survive. A low selection pressure will have a slow rate of convergence to the optimum solution and can possibly stagnate, whereas a selection pressure that is too high may get trapped at local maxima due to loss of diversity. The choice of which selection method to use is dependent on what type of problem is to be solved. For instance, proportional selection, linear ranking and tournament selection have a comparative selection pressure that increases in the order listed [14]. Genetic selection depends on a range of variables such as selection pressure, loss of diversity, bias, selection variance, selection intensity, and takeover time.

- Selective pressure: is the ratio of the probability of the best individual being selected, to the average probability of selection in all individuals within the population. It indicates the population diversity after the selection process. If the selective pressure is too small, then little or no improvement in the population fitness may occur; if the selective pressure is too high, it is possible to get premature convergence with the individuals centred on a local maximum.
- Bias: is the probability that an individual with a relatively high fitness will be retained after selection.

- Loss of diversity: reflects the reduction of diversity of the individuals within the population due to the selection process.
- Selection intensity: defines the population's expected average fitness after the selection process has been applied. A sign of loss in diversity in a population is when the population has a high average fitness.
- Selection variance: reflects the change in the population's fitness distribution compared to the normalized Gaussian distribution after the selection process.
- Take over time: is the time taken for the complete population to be replicated with the best individual.

2.4.1 Selection Schemes

There are many different selection methods each with its own advantages and disadvantages in both selection pressure, and maintenance of diversity. Some of these schemes are described in the next section.

Fitness proportionate selection

This is a probability based selection method where the chance of an individual being selected is dependent on its fitness divided by the average fitness of all individuals. An individual with a higher fitness therefore has a higher chance of being selected. This method of selection can be illustrated by spinning a roulette wheel or turning a stochastic universal sampling wheel as shown in Figure 2-14. Although individuals with a higher fitness have more numbers on the roulette wheel and thus have a better opportunity to be selected, poor individuals still retain a chance of being selected.

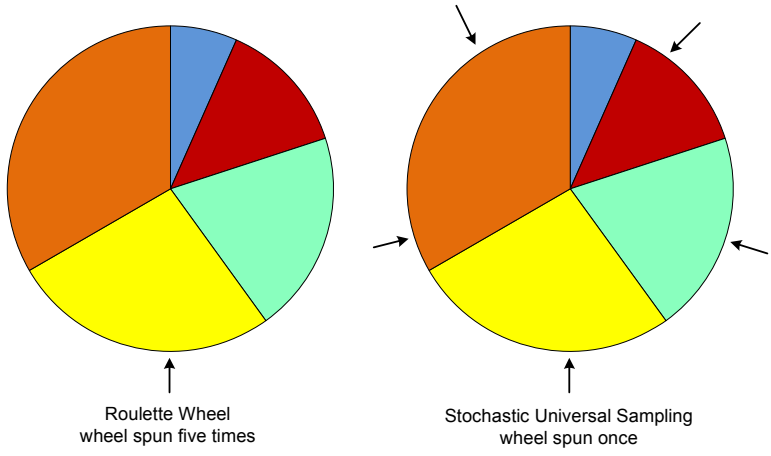


Figure 2-14. Pie graph showing the operation of roulette wheel and stochastic universal sampling selection.

In the roulette wheel sampling technique, the wheel is spun repeatedly, and an individual is chosen on each spin until the required number of offspring is achieved. In contrast, stochastic universal sampling requires only one spin of the wheel, with multiple selection points evenly spaced, with the starting point being selected at random. Fitness-proportionate selection over several generations maintains diversity in its population and produces a fast increase in fitness of the best individuals although the average fitness is slower to increase.

Rank-based selection

Rank-based selection uses a process similar to the roulette wheel; however, each individual is given a fitness rating that is dependent on its rank within the population rather than its absolute fitness. After ranking, the fitness will range from one to the population size.

All individuals will have a different rank, even if they have the same fitness level, therefore every individual will have a different probability of selection. This effectively introduces a uniform scaling, and controls selective pressures, i.e. selection pressures are reduced when the fitness variation is high and increased when the fitness variation is low, thus preventing one individual from dominating the selection process.

Rank-based selection can be linear or exponential [15]. Rank-based selection provides a simple way of controlling the selective pressure. In the pie graphs shown in Figure 2-15, one individual dominates with fitness at 50%; however after ranking, this individual has only a slightly better chance of being selected than the second or third ranked individual.

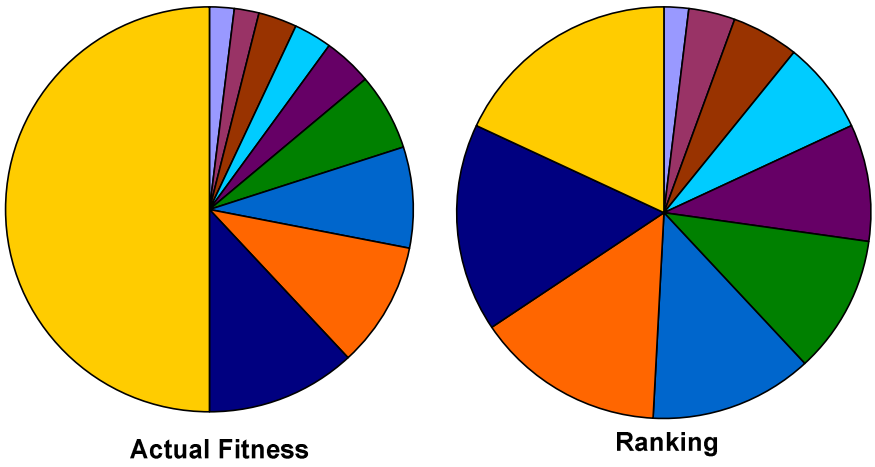


Figure 2-15. Pie graph showing the selection chance in rank based selection.

Tournament selection

This selection process divides the individuals within the population into subgroups, and within every generation the best individual in each subgroup is retained. The subgroups can vary in size from two or more individuals. The larger the subgroup size, the greater the selection pressure. With this method, the individuals with the highest fitness are generally selected; however, some low fitness individuals will also be retained maintaining diversity in the gene pool. Tournament selection is a favoured technique for hardware evolution as it is efficient, easy to implement and capable of parallelization.

Microbial Genetic Algorithms

Microbial genetic algorithms [16] is a specialized form of tournament-based selection which seeks to improve the efficiency of a standard tournament selection. Instead of generating and evaluating a new population in one step, a microbial genetic algorithm selects two individuals in a population at random. These two individuals are used to generate a new offspring which will replace one of the least fit parents within the population, as shown in Figure 2-16. This is known as a steady state, rather than a generational population.

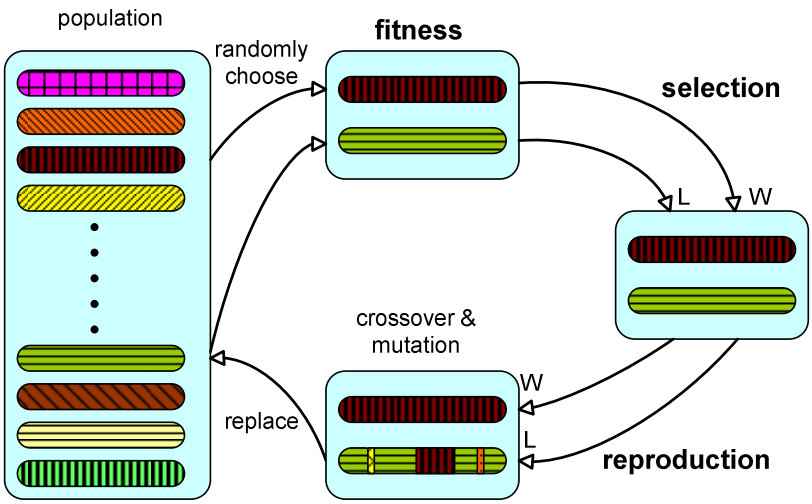


Figure 2-16. Pictorial sequence of operations in microbial selection.

This algorithm is useful for evolvable hardware as it is simple to implement. An example of its implementation was the evolution of a hardware controller by Okura et [17] for a Kephera robot for obstacle avoidance using a microbial genetic algorithm which acted on the configuration bit stream of a Xilinx XC6216 FPGA.

Elitist selection

Elitist selection allows a limited number of the fittest individuals to go through to the next generation without modification by the crossover and mutation operators. This ensures that the maximum fitness is never lost by the destruction of the best individuals. However, a high number of elite individuals will cause a loss in population diversity. Elitism is often used in conjunction with other selection schemes.

Scaling selection

Scaling selection changes its selective pressure of the population as it evolves. Initially, the selection pressure is low allowing a wide range of individuals to survive to the next generation, but as average fitness increases, the selection pressure increases reducing the number of individuals who will survive. Thus, as the average fitness of the population increases, the selection pressure increases. Scaling selection initially has a low selection pressure but as the fitness of the individuals reaches an advanced level, the selection process becomes more discriminating. This method can be used as part of the fitness proportionate schemes previously discussed, changing their selection pressure from a constant to a scaling factor. If the selection pressure is too high to start with, convergence is faster; however the population diversity is quickly lost and the process may be trapped at a local maximum. Several methods are able to determine the scaling: linear scaling, sigma scaling, windowing [18] and relative fitness [19].

Generational selection

Generational selection mimics a real life situation where no parents are retained and the best offspring are passed on to the next generation.

Steady state selection

In this process, a large percentage of the existing population will be kept after the selection process. Only a few of the best new offspring are retained to replace a small number of the worst existing population.

Hierarchical selection

Selection occurs multiple times within each generation; the first selections are simple and fast, eliminating many of the weaker offspring. The selection process then becomes more complex as the offspring numbers are reduced. This speeds up the overall selection process as the initial selection process takes less time.

Fitness Uniform Selection

Fitness uniform selection [20, 21] has a selection pressure that favours individuals in sparsely populated fitness regions, rather than directly selecting an individual with high fitness. The selection method finds the highest (f_{max}) and lowest (f_{min}) fitness in the population. A random value is generated from a uniform distribution between the values of f_{max} and f_{min} and the individual with the fitness nearest this value is selected. Fitness uniform selection will have a higher selection pressure with a population of average poor fitness, and a lower selection pressure with a population of average high fitness. The norm being a population with only a few fit individuals, however, only one fit individual is required and there is a high diversity kept in the population.

Island Model

This model is suited for parallel applications typically found in FPGA's where multiple genetic algorithms can be run simultaneously. It uses subpopulations on separate islands which are evolving in parallel, on parallel genetic algorithm machines. Periodic migration occurs where individuals are exchanged between subpopulations from the different islands. Thus, if the total population was T_{total}, and the number of islands was I_{island} the subpopulation of each island is $I_{subpopulation} = T_{total} / I_{island}$, where the total population is divided equally between islands. Two important parameters in the island model are migration size, which is the number of individuals that will be transferred when migration occurs, and migration interval, which is the number of generations that occur between migration [22].

2.5 Fitness Evaluation

2.5.1 Simulation

Using a genetic algorithm to evolve a robot controller is difficult due to the complexity of the robot's actions and how it interacts with its environment. The robotic controllers used for this process have a large search space which the genetic algorithm needs to explore and thus large numbers of generations are required before a suitable controller can be evolved. This is time consuming if performed in real time on an actual robot, and potentially damaging to the robot and its environment. To overcome this problem, evolutionary robotic genetic algorithms are normally performed using software simulation of the robot. Once a suitable solution is found, it can be transferred to the actual robot. Simulation allows a rapid increase in the speed of evolution, however,

creating a simulation for a robot means modelling the real world which can never be entirely accurate, thus the final solution will carry with it the flaws in the simulation.

2.5.2 Simulation Methods

There is a sequence of steps that are required in order to create a simulation of a robot. These are described below;

Create the simulation from the robot itself: Record empirical data of the sensors and actuators of the robot in real parts of its working environment to refine the model of the simulation. For example, Lund and Miglino [23] created a simulation of a Kephera robot by moving it through a maze, noting the activation of its sensors and also the motor settings and how they affected the actual motors. By using this simulation, a 98% reduction in time was achieved against a real life adaption. When the evolved controller was transferred to the real environment, no drop in fitness was noted.

Introduce noise into the simulation: Once the initial simulation is developed, noise can be introduced into its inputs (sensors) and outputs (actuators) to more accurately represent a real life environment. The difficulties matching a simulation to the real world are numerous. Real-world sensors often do not give accurate readings, have uncertain responses and will vary between sensors themselves, whereas actuators do not react precisely to their input signals, have response times, and other physical traits such as friction, inertia and wear. Some of these can be included in the mathematical models, however, not all discrepancies can be accounted for. Some of these discrepancies can be overcome by adding noise to the robot properties. Jakobi et al. [24] demonstrated that when the noise level in the simulation was similar to the noise level in reality, an evolved controller was more likely to work.

Validate the simulation: After the simulation has been completed, it must be tested and its responses compared to that of the real robot for validation [25]. The simulation should receive only the information that the real robot receives.

Methods to overcome the inaccuracies of simulations can be used. These include carefully modelling the parameters of the robot, taking empirical data of the input sensors and characteristics of the activators, and taking into account noise on the inputs. The problem is that a simulator can create an environment that is too clean.

There needs to be final adaptation or evolution in the real world; thus transferring the controllers to physical robots is a major challenge.

2.5.3 Types of Simulation

Simulation of robotic actions: A robot simulation can be based on a set sequence of actions that the robot must perform to obtain the desired outcomes, for instance, moving an arm to a weld joint and then activating the welder is a task a robot in a car manufacturing plant would perform.

Simulation of robotic behaviours: The robot simulation can be based on a set of behaviours which the robot may display, with a range of abilities ranging from low to high behaviour. Low level behaviours may be simple movements such as move forward, turn or move backward. These are represented as behaviours and these behaviours are evolved. A high level behaviour will involve purpose, such as interaction with the environment such as moving out of the path of an obstacle. Three processes for evolution of behaviour are: a) the primitive phase, which uses mutation to create a wide range of behaviours; b) breeding, which uses crossover to refine behaviour; and c) competition, which tests behaviours in the real world. These processes are then repeated [26].

2.5.4 Continuous Evolution

Continuous evolution is also referred to as life long adaption or punctuated anytime learning. Continuous evolution has two stages, the initial evolution stage where a controller is evolved in simulation, and a lifelong adaption stage where the controller continuously evolves as the robot is being operated. This allows the robot to adapt to a change in its environment, such as wear in an actuator. This method requires the evolution process to operate relatively quickly so that a new solution can quickly be found if a fault suddenly occurs. Although the physical robot is using the best chromosome, it still contains a population of chromosomes. The simulation and genetic algorithm are continuously running inside the actual robot modifying the chromosomes in the population. The evolution can be paused at anytime and a new best solution uploaded to the robotic controller. As long as the population continues to improve, it is possible to improve the model of the simulation. This is achieved by dynamically updating the simulation itself based on the environment and current state of the robot.

Thus, if some damage occurs to the robot, it can model this in the simulation and evolve a controller that best suits this change in the environment [27-30].

2.6 Chromosome Data Types

Robotic controller's chromosomes have a wide range of data types that a genetic algorithm can be executed on. Some of these are listed below:

- **Evolution of coefficients within a formula:** Capi et al. [31] evolved a controller that was used to provide trajectory information for a prismatic joint. They did this by generating specific X, Y co-ordinates that the robot's foot would travel through and to. The formula contained real number coefficients which were the chromosomes that were evolved.

- **Evolution of a lookup table:** This was used by the author, Beckerleg et al. [32-34], where a lookup table was evolved for a robotic controller designed to balance an inverted mobile pendulum.

- **Evolution of behaviours:** Thomaz et al. [35] used a chromosome comprised of a group of behaviours for the navigation of a robot. These behaviours were basic robotic actions such as forward, backward, left or right. The evolved controller was not locked into one environment i.e. it was able to adapt to a new obstacle course. The robot's behaviours were influenced by its current location which was determined by its infrared proximity sensors and its desired direction towards its goal.

- **Evolution of artificial neural networks:** The weighting coefficients of an artificial neural network can be evolved for robotic control. These are described in chapter four.

- **Evolution of a binary tree:** This is often used for robotic path planning [36]. In this case, the path is represented as a set of angles which are stored in a binary tree, with each node in the tree representing the angle that two path segments can take. Note a binary tree is a data structure where each node has a maximum of two possible branches. The tree itself is represented as an array of angles, with the array indices relating to the position on the binary tree as shown in Figure 2-17.

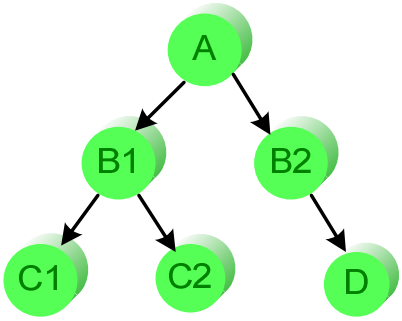


Figure 2-17. An example of a binary tree structure.

- **Evolution of integer numbers:** Ahuactzin et al. [37] used a population of individuals comprised of two eight-bit variables concatenated together into an integer to create a path plan for a robot so that it could move an arm with two degrees of freedom around obstacles. The two eight-bit variables described a particular configuration of the arm, while the sequence of these variables within the chromosome described the path that the robot was to follow.
- **Evolution of lists:** This has been used in robotic path planning [38] where each list or chromosome represents a path, beginning at the starting point and describing the path to the final point. The chromosome represents the path and obstacles location of the map that the robot is to transverse.

2.7 Subsumption

This type of robotics based on behaviour was first introduced by Brooks [39, 40]. It is a method of reducing complex behaviours into simpler layers of behaviour. Each layer can use the preceding layers, thus we can start with simple layers of a control system, and add a new layer on top in order to move to a more complex control system. Brooks started by building a complete robot controller that could achieve basic tasks (layer zero). He then built a second layer, the control layer (layer one) which could write and read data from layer zero. Layer zero ran unaware of layer one but layer one was now able to provide more complexity to the designated tasks. The same process could be repeated as the complexity increases. The interaction between layers in subsumption was either by way of passing messages, or by using suppression or inhibition methods. Brooks used the subsumption architecture in order to navigate a robot through a maze. Layer zero was designed so that the robot would not come in contact with an obstacle. If an object was in its path, the robot would move out of the way and then halt. Layer one was designed to wander around the environment, setting a new direction every 10 seconds. It is linked to Layer zero to avoid any obstacle in its path. Layer two was

designed to explore the environment using vision to find things of interest; it was linked to layer one to give new directions, and to layer zero to avoid obstacles.

Brooks expanded this research to control a six legged walking robot [41]. He used augmented finite state machines (AFSM) with registers and timers as the basis for each layer. The AFSM lower layers are connected directly to robot hardware, whereas the upper layers could write to the control registers of the AFSM below. Brooks then added levels of behaviours such as standing up, simple walking where all six legs were controlled for motion, force balancing when on uneven terrain, leg lifting to walk over obstacles, utilisation of whiskers for obstacle sensing, prowling and steered prowling for intelligent roaming.

Subsumption can be seen at a very high level such as artificial intelligence. Brooks [42] describes how this behaviour-based approach can be used to create human behaviours in robots. These human behaviours are built upon layers: 1) bodily form, where the robot morphology is in human form; 2) motivation, why the robot should react; 3) coherence, how the robot switches between tasks; and 4) self adaptation, how the robot changes its behaviour with a changing environment.

2.8 Mutation Only Genetic Algorithms

The main reproduction operator in a genetic algorithm has historically been the crossover operator, with the mutation operator used as a means of maintaining diversity in a converging population. As the chromosome is altered mostly by the crossover operator, both the mutation rate and the mutation probability are kept at a low level. In contrast, other evolutionary computational techniques such as evolutionary strategies and evolutionary programming have mutation as the driving force, thus the mutation rate and the mutation probability are at a high level.

An alternative to a genetic algorithm with both crossover and mutation is the use of a genetic algorithm with mutation only. Mutation only genetic algorithms have been used in both software and hardware genetic algorithms. The main advantage of using a mutation only genetic algorithm in hardware-based genetic algorithms is the reduction in the hardware resources of the device that implements the genetic algorithm. This is because the crossover operator is hardware intensive.

2.8.1 Software Genetic Algorithms Using Mutation Only

Siva et al. [43] used a mutation only genetic algorithm to evolve an image recognition algorithm that could identify images such as faces and chess pieces. They developed a compact genetic algorithm where only one individual was used, based on a probability vector rather than a population of individuals (compact genetic algorithms are described in chapter three). These researchers used elitism for the selection method. After fitness evaluation and selection, only one individual was kept and mutated to create new off-spring. Siva et al. [44] compared this genetic algorithm with other compact genetic algorithms which used both crossover and mutation on standard mathematical problems and found that the mutation only algorithm had a better quality of solutions and convergence speed with a smaller population base.

Zhang and Szeto [45, 46] used a mutation only genetic algorithm based on a matrix to solve the classic knapsack problem. There are several variations to this problem, with one being that there are a number of objects with a specific weight and value that we wish to place into a knapsack that is capable of carrying a maximum weight. We need to determine which objects to place in the knapsack to maximize the value that the knapsack will contain. To solve this problem, Zhang et al. used a two dimensional (N x L) matrix to store a population of individuals. The fitness and locus were also incorporated into the matrix. The matrix population was divided into three groups: parents, children and randomly generated chromosomes for population diversity. Alongside this was a mutation matrix (N x L) giving the probability for mutation. This probability could be varied depending on the fitness of the individual, with a high fitness having a smaller probability of mutation than that of a low fitness. The actual mutation rate was derived from the fitness distribution of the population, making it independent of the specific problem the genetic algorithm was solving. This evolution compared favourably with standard methods. Zhang labelled the method a ‘mutation only genetic algorithm’ (MOGA).

Shiu and Szeto [47] used a mutation only genetic algorithm to optimise the airport capacity of Beijing International airport. The mutation rate was varied depending on the fitness of the individual. A chromosome with a high fitness required a low mutation rate and was considered to be in an exploitation mode, in which a fit chromosome was exploited to optimize its performance. A chromosome with a low fitness required a high mutation rate and was in an exploration mode, where the high mutation rate allowed a

rapid examination in other areas of the search space. By making use of the locus statistics, the worst genes had a higher probability to mutate, thus the mutation rate was self adaptive.

Xia et al. [48] used a mutation only genetic algorithm to search for successful investment strategies in a stock market, using the yield of the investment over a fixed period to determine the fitness. The chromosome was a set of rules about whether to buy, sell, hold or swap a stock relative to the moving averages of that stock quoted in the stock market. Historic data from the NASDAQ was used to evaluate the outcome of the investment. It was found that the genetic algorithm gave a higher overall return, bench marked against investing in one stock.

Aguirre and Tanaka [49] investigated the effects of selection mutation and drift on genetic algorithms on NK fitness landscapes, where N is the number of genes and K reflects the interactions between genes. The NK fitness landscape relates the interdependency of the genes within the chromosome to the chromosome fitness level. The researchers found, under certain values of K, that a mutation only reproductive operator performed similarly or better than standard genetic algorithms with crossover.

Bäck [50] investigated the evolution of bit strings with a mutation only algorithm for a range of coding problems. This researcher found that a constant mutation rate of 1/L (where L is the bit string length), was sufficient for unimodal functions with only one maxima. However, when the function became multimodal with multiple local maxima, a variable mutation rate produced a better result.

Schaffer et al. [51] developed an evolutionary process based on mutation only called Naïve Evolution. They compared this algorithm with that of a full genetic algorithm which incorporated crossover and mutation, using mathematical functions as a testbed. Although the performance of the mutation only genetic algorithm was not as good as that achieved by a full genetic algorithm, it did perform well.

Lau and Tsang [52] created a mutation only genetic algorithm to solve a processor network configuration problem. The task was to network a group of computers each with a limited number of communication channels. The fitness criteria were based on the chromosome which had the shortest maximum distance between processors. The crossover operator was not used as it was likely to damage the chromosome with an incorrect routing which would have to be repaired at every reproduction stage, and it

would also significantly increase the time taken for the genetic algorithm to operate. However the mutation only algorithm was designed so that no genetic damage would occur. A comparison of the mutation only and a full genetic algorithm was made, and it was found that the speed of the evolution process matched that of a conventional genetic algorithm process.

Falco et al. [53] investigated mutation only reproduction operators, focusing on two nature based mutations, frame-shift and translocation. These were considered to be more powerful than normal point mutation operators. The frame-shift operator was based on the actual biological mutation of the nucleotides in a DNA sequence. The mutation modified a block of the chromosome and can operate in two modes, delete-first or insert-first. The operation is explained with the aid of an eight bit parent 11001001 as shown in Figure 2-18.

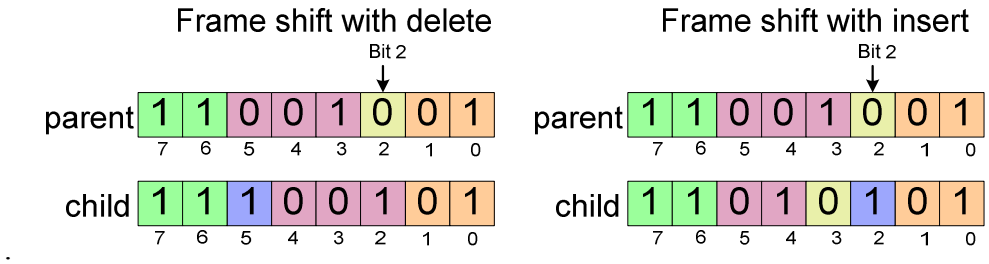


Figure 2-18. An example of a chromosome after frame shift mutation.

In this example, a random point in the parent chromosome is picked (bit 2), and a randomly chosen block size of three (bits 3-5) is also selected. In the delete-first mode, bits 0-1 are directly copied to the offspring (child), and the parent bits 3-5 are copied to offspring bits 2-4. Bit 5 of the offspring is filled with a random value (1), and bits 6-7 of the offspring are copied directly from the parents. The end result of this process is an offspring consisting of 11100101. In the insert-first mode, bits 0-1 are directly copied to the offspring, while bit 2 is given a random value, for example 1. Bits 2-4 of the parent are then inserted into bits 3-5 of the offspring and bits 6-7 are directly copied giving an offspring 11010102.

The translocation operator divides the chromosome into blocks and transposes a segment of a block with that of a second segment and block as shown in Figure 2-19.

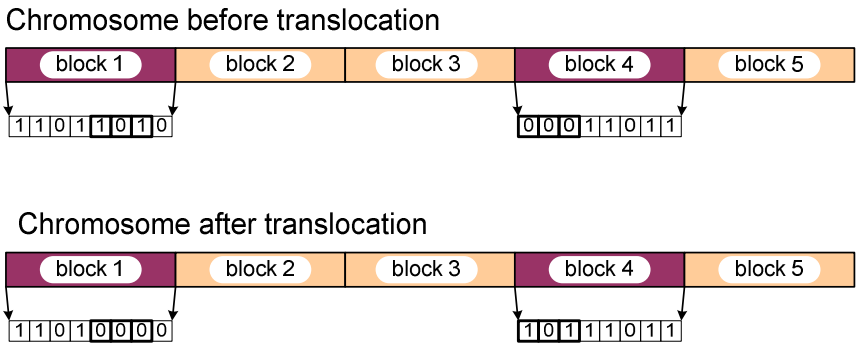


Figure 2-19. An example of a chromosome after translocation mutation.

Falco et al. tested these two mutation forms on a wide range of classical functions and found that although mutation only was not as strong as a classical genetic algorithm with crossover and mutation, it still produced significantly good results.

2.8.2 Hardware Genetic Algorithm Using Mutation Only

Nadav et al. [54] repeated the classic tone discrimination experiment undertaken by Thompson et al. [55], using a FPGA with a fine grained architecture for comparative studies of various genetic algorithm parameters. They compared the standard genetic algorithm using crossover and mutation, with a mutation only reproduction operator. They argued that crossover was important in the first stages of evolution when the chromosome were more widely spread, but less important as the population diversity diminished. The population size was varied ranging from 500, 50, 5 and 1 with the results shown in Table 2-3. With a population size of only one individual, the selection process still allowed the fitness to evolve. It was noted that the fitness showed most improvement in the later stages of the evolutionary process.

Population size	Generations	Circuits tested
1	68,000	68,000
5	28,000	140,000
50	5,000	250,000
500	fitness not reached	500,000

Table 2-3. The relationship between population size and the number of circuits tested before a successful evolution was achieved.

From these experiments, the researchers found that the evolution of a population of one using single-point random mutation alone required the testing of less than half the circuits used in larger populations with the use of crossover.

Zhu et al. [56] created a hardware genetic algorithm called an optimal monogenetic algorithm that only required a population of two individuals and used only the mutation

operator. They used two interactive search processes to investigate the search space; a global search using overlapping regions that were capable of searching through the entire search space, and a local search that would examine an area of interest. To achieve this, the genetic algorithm selected the best individual within the entire search space that had been searched by a local search to date. The second individual selected was the best in the local search currently being performed.

The optimal monogenetic algorithm randomly generates individuals, and evaluates and records their fitness replacing the individual if their fitness improves. If the fitness has not changed over n generations, a local search is performed using the local best individual with a low mutation rate. The local individual will then be replaced if a better candidate is found. After n generations with no improvement, the local search is stopped, and the local individual will replace the global individual if the fitness is better. This process is repeated until there is no variation in the global individual fitness after n reiterations. This technique performs well in comparison to the compact genetic algorithm using a hardware roulette system, when tested on algorithms that would simulate a real world search problem.

Sekanina et al. [57] created two mutation only hardware genetic algorithms. The first was used to evolve gate level multifunctional combinational circuits such as a multiplier-sorting network. Multifunctional circuits were designed to alter their functions depending on non-logic variables such as supply voltage and temperature. Standard methods of logic synthesis were not designed to include these variables; however, evolved hardware was affected by variations in voltage and temperature and thus could be used in the generation of multifunctional circuits. The genetic algorithm had a population size of fifteen and the mutation operator was bit inversion. A linear feedback shift register seeded from the computer was used to generate the population. The selection process was in steady state, which meant that if the mutated offspring had a better fitness than its parent, then the parent would be replaced by the offspring.

The second mutation only hardware genetic algorithm designed by Sekanina et al. [58] was used to evolve three bit multipliers, adders, multiplexers and parity encoders. The genetic algorithm initially generated 1024 random individuals which were evaluated for fitness with the best four retained to form the starting population. The genetic process mutated each chromosome within the population, and if the resultant offspring was better than the parent, the offspring would replace the parent.

2.9 Evolutionary Robotics and Lookup Tables

Lookup tables have been used in evolutionary computation in a variety of applications such as robotic simulation, cellular automata and FPGA functional elements. However, to the author's knowledge, not as a robotic controller. In conjunction with this thesis, the author along with other staff at AUT university investigated the use of evolving lookup tables for generating walking gaits for a hexapod robot [59]. The hexapod motion was controlled by eighteen servo motors (three on each leg). A two-dimensional lookup table (nine by eighteen), as shown in Figure 2-20, described the gait sequence of the hexapod. There were nine discrete gait stances linked to the rows, with each gait stance detailing eighteen motor angular positions, one for each servo motor on the hexapod robot. There were twenty motor angular positions ranging from $\pm 45^0$. A simulation of the hexapod was performed in Matlab based on a physical robot constructed from a Lynxmotion kitset with each leg having three degrees of freedom (pelvic joint, hip joint and knee joint). The fitness was calculated from three conditions: the robot walking forward in a straight line, the stability of the body of the robot, and the efficiency of the motion, (using the least number of steps). The genetic algorithm had a population of 100 lookup tables; the reproduction used two-point crossover with a mutation rate of 0.31 percent and the selection method was tournament. It was found that a successful gait was evolved within 700 generations.

motor angles

gaits	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	26	17	18
1	45	25	5	-10	5	-25	10	10	30	25	-20	15	25	10	25	15	15	-15
2	35	-40	-25	0	20	-15	-25	-10	0	-30	-25	20	-25	0	-20	-25	20	10
3	0	45	15	0	15	-5	10	25	25	-40	-10	0	-30	25	-40	-25	0	10
4	-15	-10	25	-45	-30	0	10	5	20	10	-25	-10	-30	-20	45	25	-5	-20
5	0	30	-30	10	-30	25	15	15	5	15	20	0	-45	5	20	20	0	-15
6	25	5	-5	15	20	25	-35	20	0	20	-10	-15	45	0	30	10	15	20
7	-45	0	-5	-40	15	5	-45	25	15	15	10	-10	30	-15	25	10	-25	-5
8	-5	-15	15	0	10	25	-40	25	25	25	-10	-5	5	-25	25	-5	5	25
9	30	25	-25	40	-5	-40	0	-25	-15	0	-15	0	-25	15	0	-40	-15	0

Figure 2-20. The lookup table chromosome for the gait of a hexapod robot.

Lund and Hallam [60] evolved a neuro-controller for a Khepera robot which was required to explore its environment and then return home to a light source. The simulation of the Khepera robot was implemented as a lookup table of possible sensor and motor responses. The lookup table was built from experimentation by placing the

robot at set positions in its environment and rotating the robot 360 degrees. The response from the light meter was noted and put into the lookup table. A second application of using a lookup table as a simulation was performed by Lund et al. [61] to co-evolve both robotic shapes (morphology), and their associated control systems based on LEGO robot parts. The simulation of the robot and its parts were included in the lookup table.

Chavoya et al. [62] used a genetic algorithm to evolve cellular automata which were used to create predefined two dimensional and three dimensional shapes. The chromosome evolved by the genetic algorithm was the cellular automata rules which defined how the cells would grow. These rules were incorporated in a lookup table and the fitness defined by how well the final structure of the cells represented the desired shape.

Greenfield [63] used actual DNA sequences from animals and plants which were converted into control sequences in order to drive a robotic simulation used to draw motifs. Two lookup tables were used for the chromosome, one that assigned codons (A, C, T, G) to robot commands, the second to assign codons that served as arguments to these commands. The fitness of each chromosome was evaluated from the motion of the simulated robot and the drawings that were produced.

Krohling et al. [64] evolved the lookup table within a FPGA's functional element. The FPGA was used to control a Kephera robot for navigation and obstacle avoidance. To overcome destructive architectures, the routing was kept fixed with only the lookup table evolved. The researchers used JBits, (a set of JAVA classes), which enabled the researchers to alter the functional elements lookup table without altering the routing, thus preventing destructive architectures.

This chapter has summarised the concepts of genetic algorithms, detailing the algorithmic process and how it can be applied to robotic controllers, in particular, the use of mutation only genetic algorithms was explored.

Chapter 3

Chapter 3: A Review of Hardware Controllers and their use in Evolutionary Robotics

Hardware controllers and hardware genetic algorithms have been used extensively in this research. This chapter presents an overview of this topic. Evolutionary computation for robotic controllers has been used to evolve software controllers, topology and weightings of artificial neural network controllers, and class structures within fuzzy logic controllers [65, 66]. It has also been used to evolve digital and analogue circuits for robotic controllers using field programmable gate arrays (FPGA) [67] and field programmable analogue array (FPAA) devices [68]. FPGAs began as simple logic devices used for interfacing between integrated circuits, but now they have sufficient resources to create complex circuits including processors and high speed switching circuits. As the configuration of circuits within a FPGA can be easily and quickly altered, evolutionary computation can be used to modify them and thus the circuit (hardware) itself evolves. In comparison with software evolution where the chromosome is a set of control parameters, the chromosome in hardware evolution (evolvable hardware) is the bit sequence used to configure the FPGA.

3.1 Commercial FPGA and FPAA Architectures

This section details common architectures used inside FPGA and FPAA devices, and explains the concepts of fine and coarse grained architectures.

3.1.1 Field Programmable Gate Array

A FPGA is a silicon device that can be configured with custom designed digital circuits. An advanced FPGA consists of several systems, such as logic arrays, memory blocks, phase-locked-loops and embedded multipliers, with some devices including digital signal processing (DSP) blocks. These systems are connected together by a programmable routing system that can be altered to configure the system according to

the user circuit requirements. Some FPGAs are hybrids, with a separate processor and a FPGA included on the same chip.

An overview of the FPGA is provided in the following section based on the Altera FPGA architecture. The heart of the FPGA is the logic element which is a RAM-based lookup table as shown in Figure 3-1. The lookup table has four inputs and one output which can be programmed to provide any logic expression of the four input variables as well as being used for storage. There are also control, clock and feedback registers which allow the output of the lookup table to be fed back to the input (d3-fb).

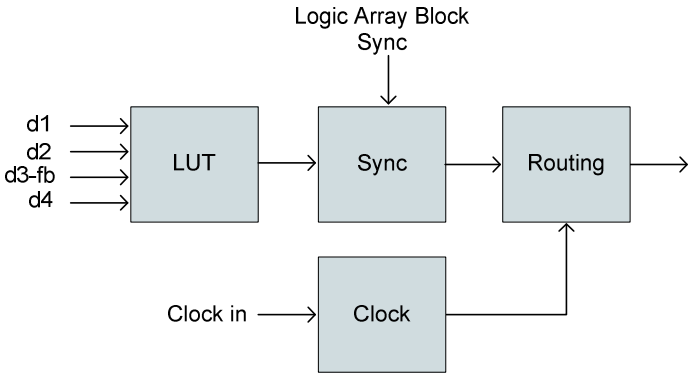


Figure 3-1. The Altera FPGA logic element.

A typical number of sixteen logic elements are grouped together to make a logic array block as shown in Figure 3-2. Xilinx terminology for this structure is a configurable logic block. This block has programmable interconnect lines between the logic elements and control logic within the logic array block.

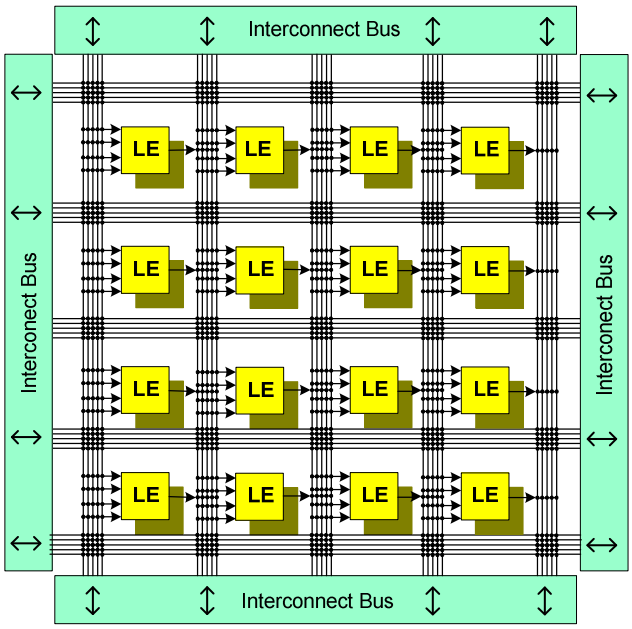


Figure 3-2. The Altera FPGA logic array block.

The logic array blocks themselves are formed into a two dimensional array consisting of hundreds of blocks. The inputs and outputs of these blocks can be routed to other blocks to create more complicated designs. These blocks can also be connected to the external pins of the FPGA device. The function of each logic element, logic array block and the way in which they are connected to other blocks is defined by a configuration bit stream that is either downloaded into the FPGA from an external computer, or loaded into the FPGA from onboard flash memory when the circuit is first powered up.

An Integrated Development Environment (IDE) such as Altera's Quartus design suite is used to create the circuit. The IDE allows the use of logic schematics, or a hardware description language such as Verilog or VHDL, to describe the circuit. The IDE compiles the designed circuit into a configuration bit stream which is then downloaded into the FPGA. At this point, the FPGA circuits are configured with the custom designed digital circuits.

FPGA devices can be broken into two classes, partial and non-partial reconfiguration. Non-partial reconfiguration will reconfigure all the FPGA logic array blocks within the FPGA each time it is programmed. This feature is common to all FPGAs. Partial reconfiguration allows sections of the FPGA logic array blocks to be reconfigured, while still retaining other logic array block configuration hardware structures. For example, a processor could be retained while other hardware structures are changed. This allows dynamic reconfiguration of the FPGA while it is operational.

3.1.2 Coarse and Fine Grained Architecture

The two types of FPGA architectures, fine grained and coarse grained are related to the granularity of the logic modules. A fine grained architecture has very simple logic blocks with a great deal of flexibility but requires a large amount of silicon resources especially in the increased routing requirements. A coarse grained architecture has very large logic modules with sometimes two or more lookup tables within the one logic element. While this requires less routing between logic elements and logic array blocks, it can be less efficient in its use of resources than a fine grained architecture.

More powerful FPGAs have predetermined sections of hardware such as multipliers, memory, floating point units, digital signal processor blocks and processors. This architecture is more efficient in power and silicon resources but less flexible. These more complex blocks are normally included within a fine grained FPGA with a great

deal of work involved in the interface between the local array blocks and the fixed hardware. Yu et al. [69] showed that fine grained and course grained architectures can also be successfully combined.

3.1.3 Field Programmable Analogue Array

The Field Programmable Analogue Array (FPAA) allows analogue operational amplifiers to be configured into a range of analogue circuits including amplifiers and filters. They are often used for analogue filtering as the frequency response of the filter can be dynamically altered. The FPAA has the same two dimensional array architecture as the FPGA; however the FPAA contains configurable analogue blocks rather than the logic array blocks of a FPGA. Both these configurable analogue blocks and the routing between them are analogue based, as shown in Figure 3-3. The configurable analogue block has a set of operational amplifiers whose gain and frequency response can be dynamically altered.

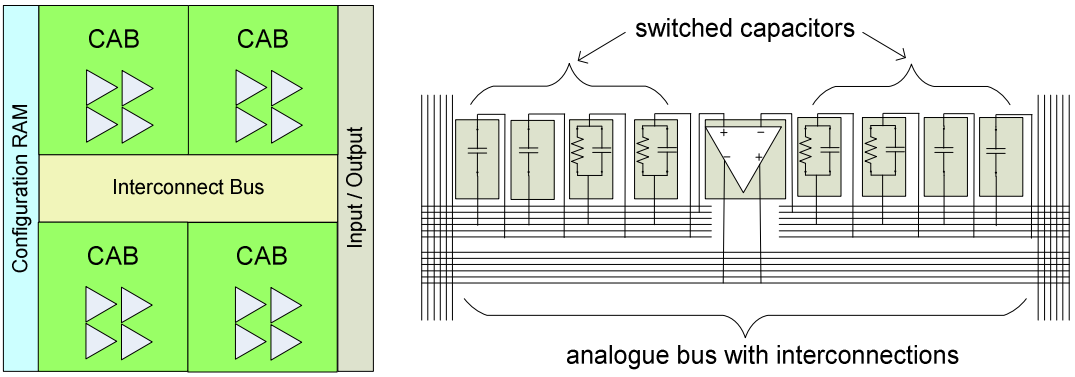


Figure 3-3. The architecture of a FPAA.

There are two types of FPAAs: switched capacitor (discrete time domain) and transconductor (continuous time domain). The switched capacitor uses the principle of creating an equivalent variable resistance by altering the frequency of a switched capacitance. It has the advantages of programmability and insensitivity to resistance of programming switches but is limited in bandwidth. The transconductor consists of an operational amplifier and programmable capacitors linked by a transconductor based array. It has the advantage of greater bandwidth but a reduced programming range for its parameters. Each configurable analogue block can be configured for different applications including filtering, addition and multiplication. Once again, the analogue blocks and routing are configured by a configuration bit stream.

3.2 Overview of Hardware Evolution

The FPGA is configured with a configuration bit stream. This configuration bit stream describes the electronic circuit and required routing that is to be created inside the FPGA. In comparison with software evolutionary computation where the chromosome is a possible solution in the form of a parameter, the chromosome in evolvable hardware is the configuration bit stream and the phenotype of the chromosome is the circuit described by the configuration bit stream.

Using standard genetic algorithm, the FPGA configuration bit stream can be evolved and then downloaded into the FPGA as shown in Figure 3-4. The ensuing circuit can be tested for fitness, with the evolutionary process repeated until a suitable result is achieved. Thus, the hardware itself evolves. Evolving electronic circuits, often referred to as evolvable hardware, has advanced from the generation of simple circuits through to more complex functional systems such as robotic navigation [67, 70].

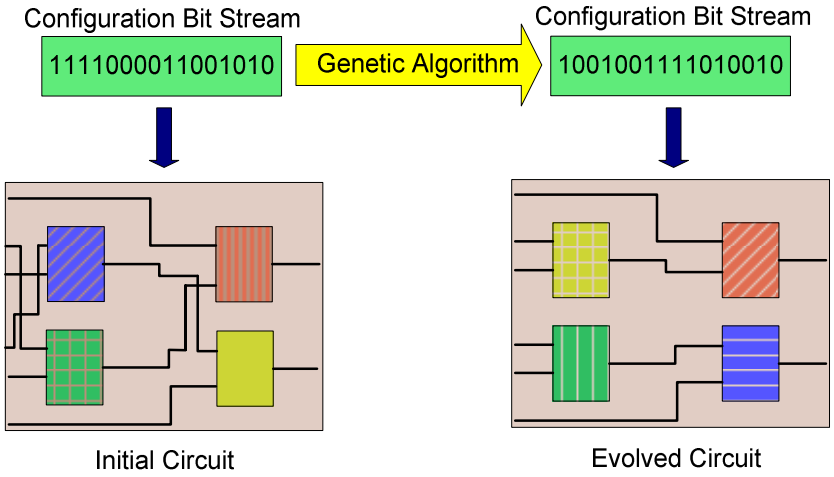


Figure 3-4. An evolved bitstream with corresponding circuit.

Historically, evolvable hardware has been broken into two methods, extrinsic (off-line) and intrinsic (on-line) evolution as shown in Figure 3-5. The extrinsic method uses a circuit simulation to test the fitness of the individual during the evolutionary process. When the evolutionary process is completed and the solution is found, the resulting best individual is then downloaded to the FPGA. The intrinsic method does not use circuit simulation to test the individual; instead the individuals are tested for fitness on the FPGA in the environment they will run in. This overcomes the problems of matching a circuit simulation to real life. The evolutionary process is still performed in software by a genetic algorithm.

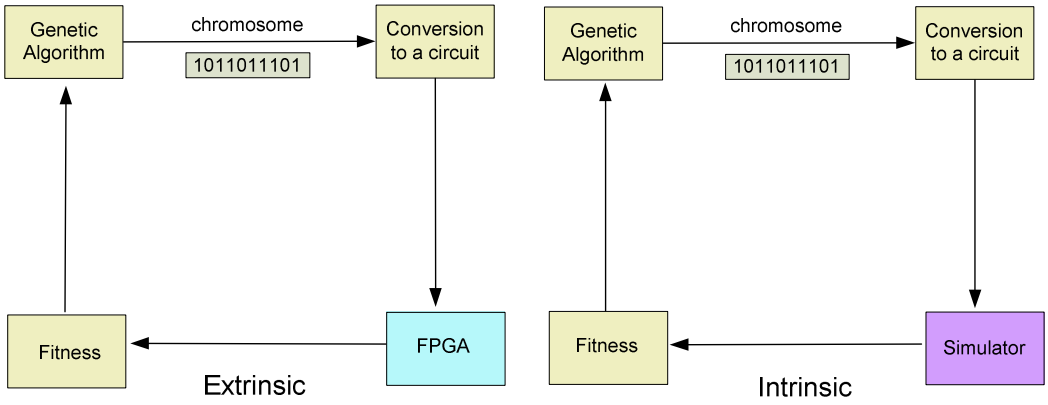


Figure 3-5. Block diagram of the extrinsic and intrinsic evolutionary process.

The first implementation of evolvable hardware was performed by Thompson [71, 72] when he created a tone discriminator using evolutionary techniques on a Xilinx XC6216. Thompson evolved a 1800 bit configuration bit stream using elitism with rank based selection, and used both crossover and mutation operators. The experiment fed two tones into the FPGA (1 kHz and 10 kHz) and the evolved circuit was able to differentiate between the two tones. However, the evolved circuit would cease to operate if variations in temperature or voltage occurred, or when the FPGA was replaced. After careful analysis, it was found that the evolved circuit did not take into effect the timing and propagation delays of the gates within the FPGA. These characteristics can be affected by changes in the temperature, voltage and device, and would normally be taken into account if an engineering design approach had been used.

Direct evolution of the Xilinx configuration bitstream was possible because the architecture of the Xilinx XC6216 FPGA made it impossible for a destructive configuration of outputs connecting to outputs to occur. As shown in Figure 3-6, the logic element has connections from the north, south, east and west.

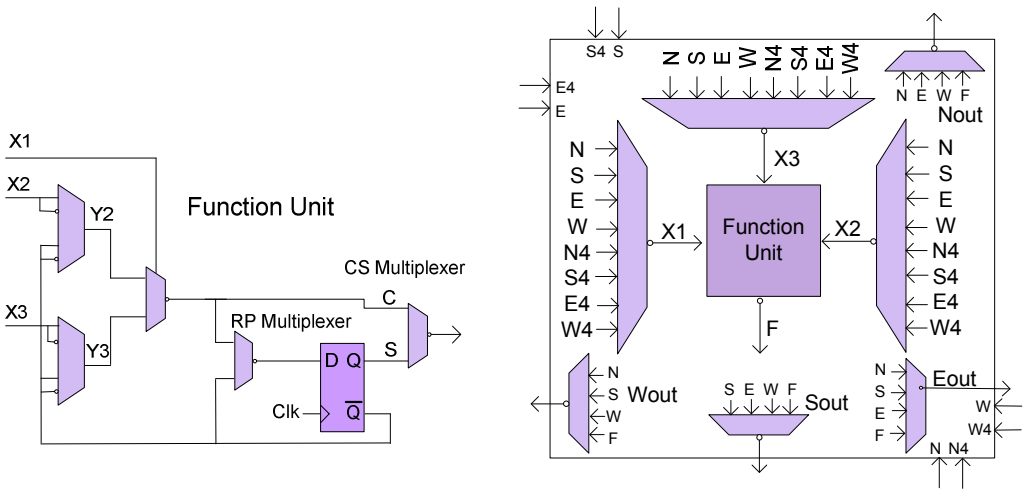


Figure 3-6. The Xilinx XC6216 logic element.

This non destructive architecture is more clearly seen in Figure 3-7 where the interconnection between logic elements is illustrated. The routing is very limited and it simply connects adjacent logic elements together in a grid like pattern. The connections between inputs and outputs are fixed, and are not affected by the configuration bit stream, thus a configuration bit stream modified by the genetic algorithm will not produce a destructive configuration. This allows the device to be directly used in evolvable hardware. However, as the XC6216 has been discontinued, it is no longer used in current research.

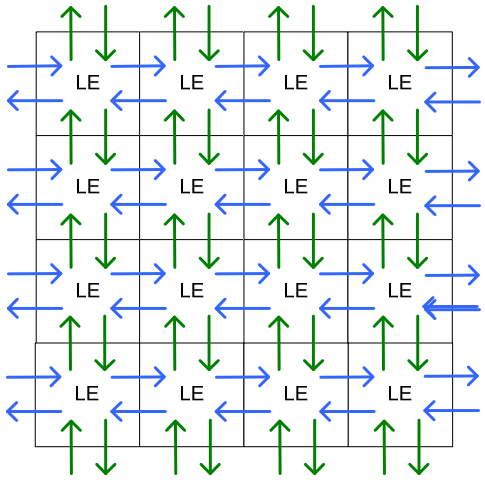


Figure 3-7. The Xilinx XC6200 logic element interconnections.

From the initial research of Thompson, and subsequent research of others, three major problems confronting evolvable hardware were found. These are: a) destructive configurations, as new FPGA devices allow output to output connections; b) partial reconfiguration, as not all FPGA devices have partial reconfiguration which is useful for

evolvable hardware; and c) a large search space, the evolutionary search space of the configuration bit level is too large to allow complex circuits to evolve.

3.3 Problems with Commercial FPGAs and Evolvable Hardware

The obstacles to evolvable hardware with modern FPGAs are: a) scalability as the search space rapidly becomes too large; b) partial configuration, as altering only part of the FPGA is not fully supported by all FPGA manufacturers; and c) destructive architectures, as FPGAs have the ability to connect outputs to outputs.

3.3.1 Scalability

Scalability is the ability of the evolutionary process to perform acceptably as the complexity of the problem increases. A FPGA with a fine grained architecture requires a large configuration bit stream. Thus, the chromosome which is being used to describe a complex circuit design will be large, and will have a large search space which limits the ability for the evolution to find practical solutions. Methods to reduce this search space and manage the scalability problem are evaluated later in this thesis.

3.3.2 Partial Reconfiguration

Partial reconfiguration is the ability of the FPGA to reconfigure parts of its circuits while the remaining sections are still running. This process is very important for hardware evolution as the genetic algorithm itself may reside within the FPGA, and should not be modified. This feature of partial reconfiguration is not often required by industry and only a few manufacturers such as Xilinx supply these devices.

3.3.3 Destructive Architectures

Modern FPGAs have complex routing systems that will allow outputs to be connected to outputs, thus incorrect routing will quickly cause permanent damage to a FPGA. In a normal development environment using a compiler such as Altera's Quartus, illegal or destructive configurations would create errors in the compilation stage, the configuration bit stream would not be generated, and the compiler would show a list of illegal operations. However, if the evolutionary process of crossover and mutation is applied directly to the configuration bit stream, then a circuit may be generated that connects an output directly to another output, therefore causing damage to the FPGA. Previously researchers in evolvable hardware used the Xilinx 6200 series which has an

internal architecture that makes it impossible for the routing to be programmed into a destructive state. This series of FPGA, however, has been discontinued.

3.4 Solutions to Commercial FPGA and Evolvable Hardware

Three methods have been developed to overcome the problems described in the previous section. The first method is to use a genetic compiler that can determine if an illegal configuration bit stream has been generated during the evolutionary process. The second method is to use genetic programming rather than genetic algorithms to evolve a hardware descriptive language (such as VHDL or Verilog) and to then use a compiler to create the bit stream. The third method is to create a virtual FPGA whose architecture will not allow output to output connections.

3.4.1 Genetic Compilers

For commercial reasons the two major manufacturers of FPGA devices, Xilinx and Altera, have not made public the configuration bit stream parameters for their devices. However two Xilinx employees, Levi and Guccione [73], have created a Java based program called GeneticFPGA, which can be used for evolvable hardware applications. As shown in Figure 3-8, this program performs the evolutionary techniques of reproduction and selection directly on the configuration bit stream, but filters out illegal or unreliable bit stream parameters generated by evolutionary techniques before downloading the configuration bit stream to standard Xilinx devices.

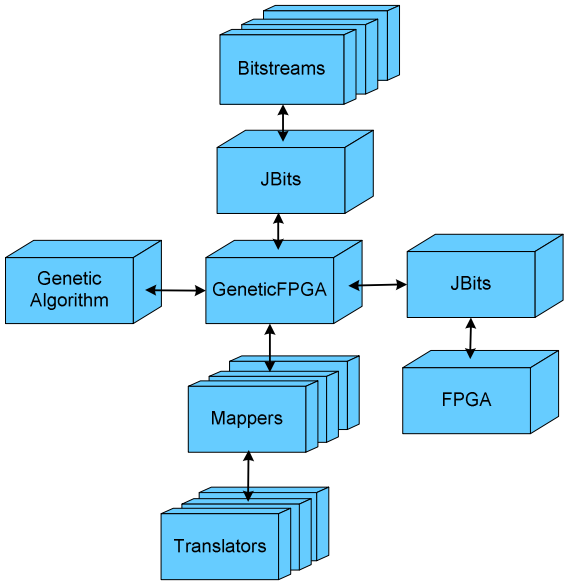


Figure 3-8. Block diagram of the evolvable hardware process using the Xilinx genetic FPGA.

JBits is a Java-based program (Application Programming Interface) produced by Xilinx, that allows the user to modify the configuration bit stream generated by the Xilinx design tools for the Virtex family of devices. This enables the user to dynamically reconfigure parts of the circuit inside the FPGA while the device is running. JBits converts the configuration bit stream into a two dimensional array of configurable logic blocks, and allows the alteration of these blocks as well as the routing between them.

Hollingworth et al. [74] used Jbits to evolve a simple adder by first creating a four by two array of lookup tables with a fixed feedforward routing between lookup tables as shown in Figure 3-9. The position of the lookup table within the Virtex configurable logic block is known, thus JBits can be used with a genetic algorithm to modify these lookup tables without altering the routing.

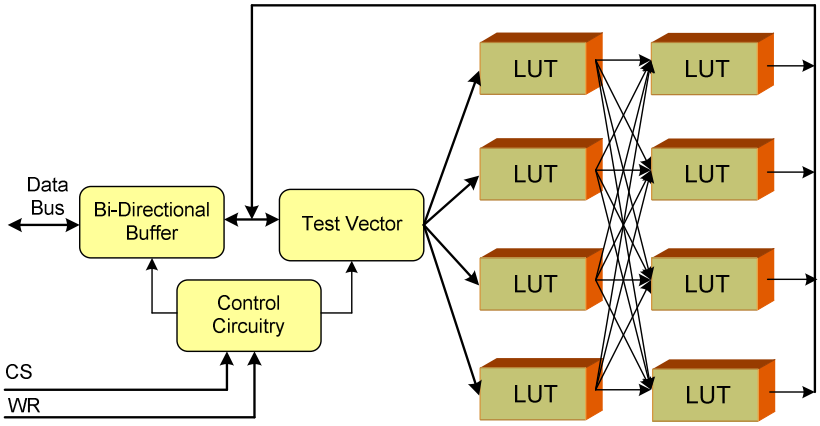


Figure 3-9. The Virtex configurable logic blocks showing the interconnections of the lookup tables.

The process for evolution is shown in Figure 3-10. The initial circuit with lookup tables and routing between them is performed in a hardware description language, then compiled and fed into the JBits program. The JBits can operate on, and modify, configuration bit streams either generated by the compiler, or read back from the FPGA. As the Xilinx FPGA allows for partial reconfiguration, it is possible to modify the contents of the lookup table without modifying the routing between tables. The genetic algorithm can then be used on the lookup table to evolve a solution.

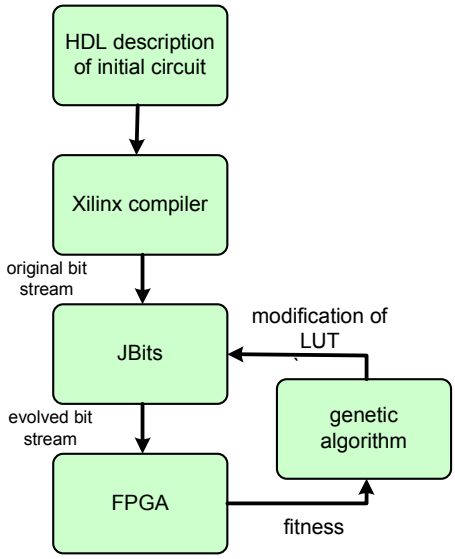


Figure 3-10. Block diagram of the genetic process using Jbits.

3.4.2 Genetic Programming of Hardware Descriptive Languages

Genetic programming evolves the hardware descriptive language program rather than the configuration bit-stream as performed by genetic algorithms. Genetic programming is normally used on a software program and will typically use a tree like structure for its evolutionary process. This is difficult to implement in a hardware descriptive language as it is not described in a similar fashion to a software language with a tree like structure. To overcome this, a parse tree representing the code is used; this is translated into a hardware descriptive code and then compiled. The parse tree is comprised of branches and terminating nodes, with the branches performing decision processes relating to the input state of the system, and the nodes relating to the desired robotic actions as shown in Figure 3-11. It is the parse tree itself that is evolved by the genetic algorithm.

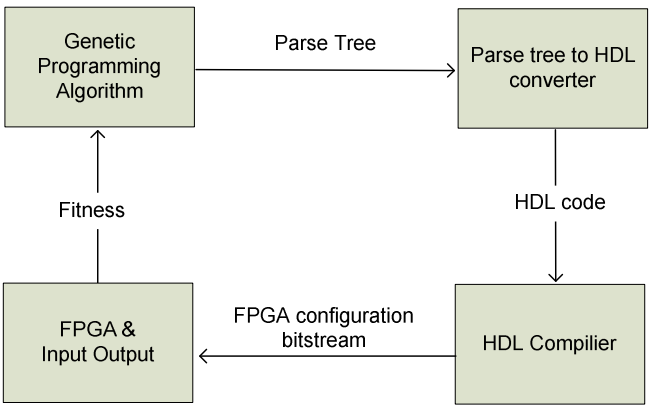


Figure 3-11. Genetic programming using parse tree.

The advantages of genetic programming are portability of the language between different devices and manufacturers, and scalability for larger problems, giving a reduced search space as evolution is occurring at a higher function level than the configuration bit stream. The problem that arises with this method is that the compilation time for anything other than simple systems can be several minutes, thus the evolutionary process becomes time consuming and inefficient. In addition, the encoding of the program so that it can be modified by the genetic program is important. Several authors as included below have suggested ways to overcome these problems.

Seok et al. [67] used genetic programming to evolve the motion of a robot towards a light source using a linear chromosome to represent the tree structure of the programme. This was performed by representing the tree as a binary string, and separating each path from the root node to the terminal node. The tree itself related the light sensors range and direction to the robot's movement, while the terminal nodes represented motor states such as forward and backward.

Dong-Wook et al. [75] used genetic programming to evolve a robotic controller. Due to the difficulties of genetic programming on a hardware descriptive language, they instead modelled the hardware on behaviours, using two sets. The first set was the function set (decision) for example if-obj, if-goal, the second set was the terminal set (action) such as move-forward, turn-left then move-forward. Dong-Wook et al. called these behaviours 'context switchable identity blocks', and used them to create a tree like structure as shown in Figure 3-12.

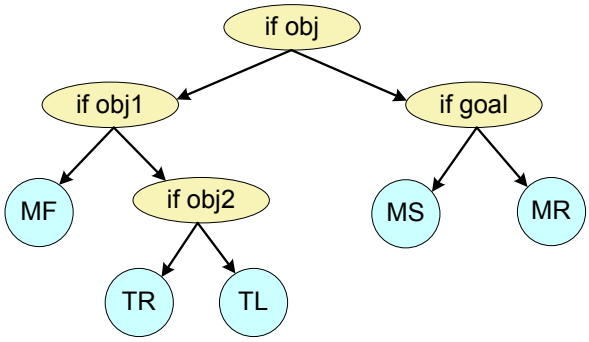


Figure 3-12. Genetic programming using a tree structure of context switchable identity blocks.

This tree is suitable for crossover operators. It was found that the tree had to be deep enough so that all possible input patterns could be obtained. Dong-Wook et al. used this system to evolve a Kephera robot to move boxes to the edge of an enclosure whilst avoiding obstacles.

Mizoguchi et al. [76] developed a program called production genetic algorithms that would allow evolution of grammatical language structures such as a hardware description language. They used production rules to regulate the evolutionary process and discarded any solutions that would violate these rules enabling the generation of complex hierarchical structures. The chromosome represented a tree structure making it possible to replicate grammatically correct offspring. They used standard genetic operators as well as duplication (copying functional blocks within the chromosome) and insertion (copying functional blocks from another chromosome). In this way, the chromosome could grow larger and create more complex circuits. The researchers used this technique to evolve an artificial ant that would follow a trail.

Montana et al. [77] used genetic programming to evolve edge detection in image processing which was coded using VHDL. They created a system called EvolvaWare as shown in Figure 3-13, which represented the hardware descriptive language using a parse tree as previously described. To overcome the time problems of compiling for each generation, the parse tree was evaluated by a software algorithm that could simulate the actual code. Only when a suitable solution was found, was the FPGA actually programmed.

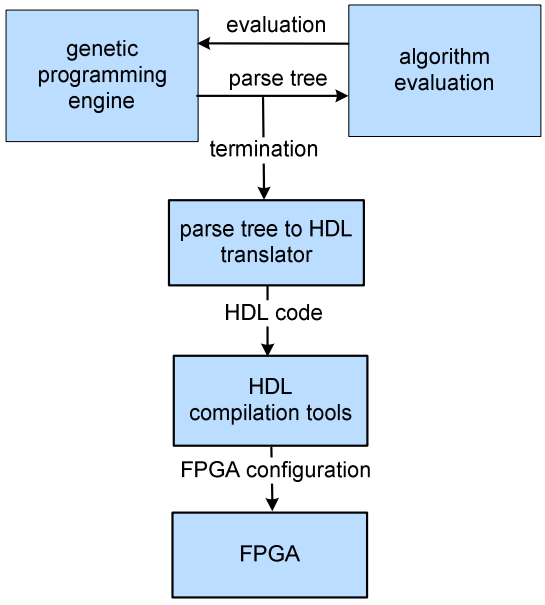


Figure 3-13. EvolvaWare structure using genetic programming and a parse tree.

Further work has been performed on Cartesian genetic programming which is a subset of genetic programming. Cartesian genetic programming was developed by Miller et al. [78, 79] where the electronic circuit can be represented as a tree-like structure and thus allows genetic programming to be used. Cartesian relates to the Cartesian co-ordinate

graph system where a specific point on a plane can be represented by a pair of numerical co-ordinates. In this case, Cartesian genetic programming uses a linear string of integers as an indexed graph to represent the program tree. In Cartesian genetic programming, the program or circuit is seen as a two dimensional array of nodes (where the node is either a programming construct or an electronic function). All the inputs and outputs to these nodes are sequentially indexed. The circuit can therefore be expressed by a genotype that shows the connections between the nodes and the functions of the nodes.

3.4.3 Virtual FPGA

The concept of a virtual FPGA is to create an 'ideal FPGA' that would reside inside a commercial FPGA. An ideal evolutionary capable FPGA would have limited routing to reduce the search space and be designed so that destructive configurations cannot occur. In addition it would have a high level or function level abstraction. Current FPGAs do not support this and are therefore unsuitable for genetic processes. However, it is possible to create a virtual FPGA with the desired features and download it into a standard commercial FPGA.

The virtual FPGA configuration bit stream which has been evolved by the genetic algorithm can be loaded in two separate ways. The virtual FPGA is created using the hardware descriptive language constructs and downloaded into the FPGA using the standard FPGA configuration bit stream interface. The virtual FPGA configuration bit stream is connected to a computer via a serial port such as a RS232 or USB serial input output system, as shown in Figure 3-14. The computer runs the genetic algorithm and fitness evaluation, downloading the new virtual FPGA configuration for each test.

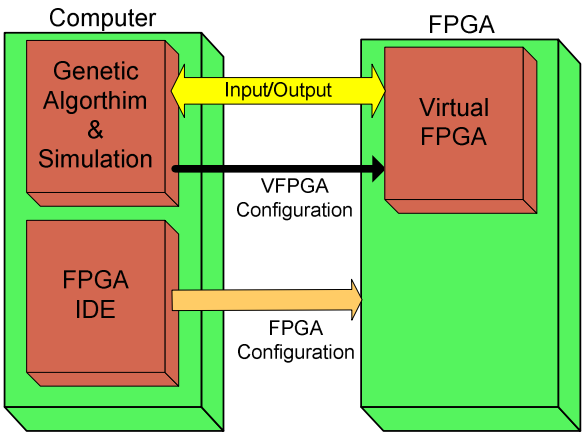


Figure 3-14. System interconnections using an external genetic algorithm with a virtual FPGA.

Alternatively, the genetic algorithm and virtual FPGA both reside in the FPGA as shown in Figure 3-15. The computer is only used for the generation of the FPGA systems, and control and monitoring of the evolutionary process. Once the virtual FPGA, genetic algorithm, and simulation are loaded into the FPGA, the evolutionary process will begin with only control and monitoring occurring on the computer. Note the genetic algorithm and simulation can be run in software on a processor inside the FPGA, or these systems themselves can be implemented in hardware.

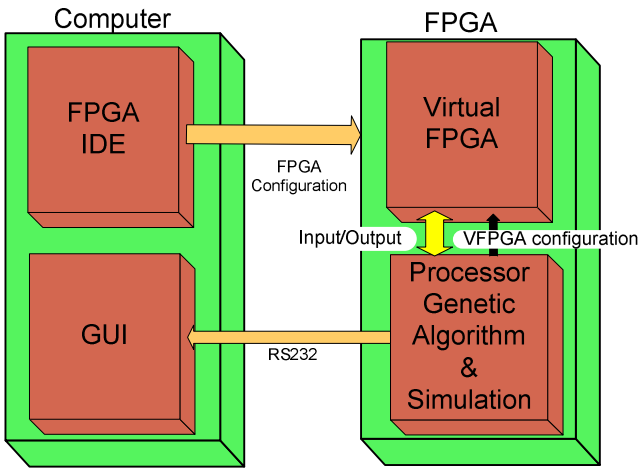


Figure 3-15. System interconnections using an internal genetic algorithm with a virtual FPGA.

3.5 Virtual FPGA Architectures

There have been several architectures suggested for this process with some researchers mimicking the Xilinx 6200 series. Other researchers have created new architectures that have a high level of abstraction and a reduction in the routing requirements making the evolvable hardware process more efficient.

3.5.1 Xilinx XC6200

The first attempt at creating a virtual FPGA was the implementation of the architecture of the original Xilinx XC6200 FPGA as shown in Figure 3-16.

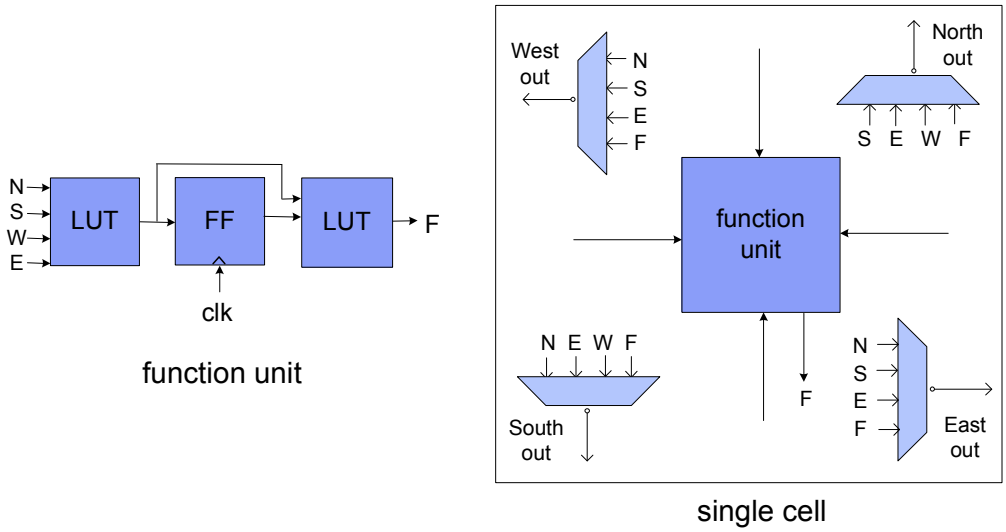


Figure 3-16. A virtual FPGA based on the Xilinx XC6200 core.

Hollingworth et al. [80] evolved digital circuits using an array of XC6200-like cells in the Xilinx Virtex. They modified the architecture so that the routing requirements were reduced, and used Jbits, (Xilinx hardware interface) to program the Virtex device. They evolved two systems: firstly, a routing puzzle where a simple connection pathway between inputs and outputs was evolved, and secondly, an oscillator puzzle, where a 55.5 kHz oscillator was evolved.

3.5.2 S-block

One of the problems in evolvable hardware on a fine grained FPGA is the large search space due to the length of the configuration bit stream that is required for the logic and routing. To help reduce this problem Haddow and Tufte [81, 82] proposed a virtual FPGA architecture that reduced the configuration bit stream by combining the logic and routing as part of the lookup table. This architecture is called the S-Block, and was designed so that it could fit into one slice of the Xilinx configurable logic block, minimizing the FPGA resources.

Inside each S-block are a lookup table and an input/output wiring structure on each side as shown in Figure 3-17. The lookup table has five inputs: one input from each of its sides (north, east, west and south), and one input feedback from its output. The output is clocked to prevent parasitic oscillations and is fed to each side of the block. Each S-block can be configured for either logic or routing. The S-block structure avoids destructive configurations as it only allows the outputs to be connected to the inputs, and vice versa.

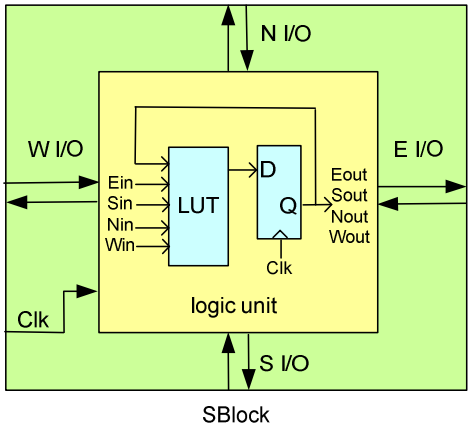


Figure 3-17. The virtual FPGA S-block showing logic unit and the S-block structure.

The S-block is laid out in a grid pattern with each block connected to the blocks adjacent to it as shown in Figure 3-18.

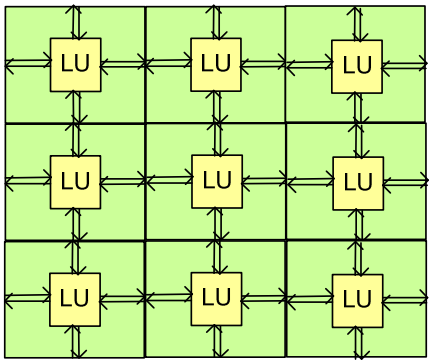
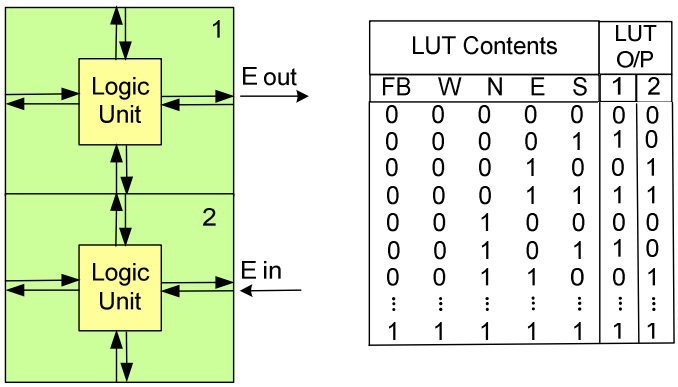


Figure 3-18. An array of S-blocks showing the interconnections between logic elements.

An example of how the S-block can be used for routing is shown in Figure 3-19. In this example a signal is fed into S-block2 from the east and then fed back out of S-block1 to the east. The lookup table in S-block2 is configured to read the input from the east, and feed this input to all its outputs. The S-block1 lookup table is configured to read the input from the south (the output of S-block2) and feed this signal to all its outputs, thus the two S-blocks are acting as a router. Other S-blocks can be configured as logic functions.



LUT Contents					LUT O/P	
FB	W	N	E	S	1	2
0	0	0	0	0	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	1
0	0	0	1	1	1	1
0	0	1	0	0	0	0
0	0	1	0	1	1	0
0	0	1	1	0	0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	1	1	1	1	1	1

Figure 3-19. The S-block lookup table contents used to rout a signal from east in to east out.

3.5.3 Gate Level and Functional Level Logic Units

Higuchi et al. [83] and Vassilev and Millar [84] showed how the functional level rather than the gate level evolution performed better by reducing the search space and thus decreasing the time taken to evolve. A gate-level evolutionary process evolves simple gates such as AND or OR gates to generate a high level circuit, as shown in Figure 3-20. This is the basic logic unit of a normal FPGA, and it allows only simple circuits to be evolved as the search space is too large for more complex circuits.

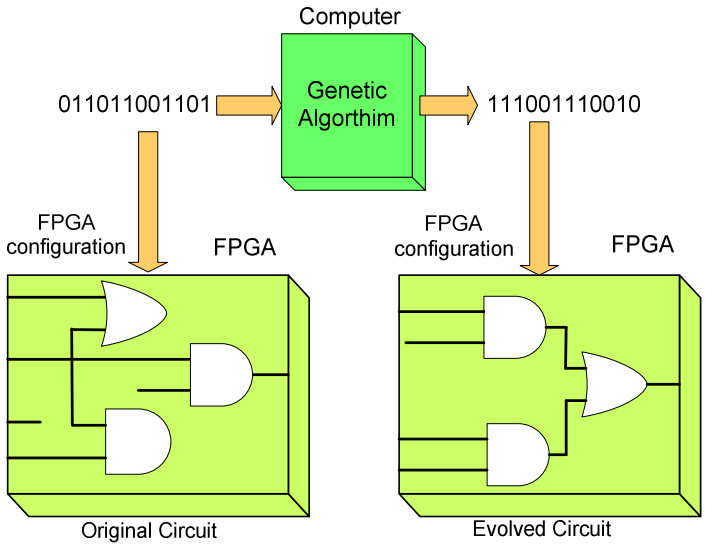


Figure 3-20. Schematic representation of gate level evolution.

However, if the circuits are comprised of higher elements or functions, more complex circuits can be evolved, as shown in Figure 3-21. The higher level functions can be arithmetic functions such as adders, subtractors, multipliers and dividers. More complex functions such as sine cosine generators, or programming structures such as if else switches can be used.

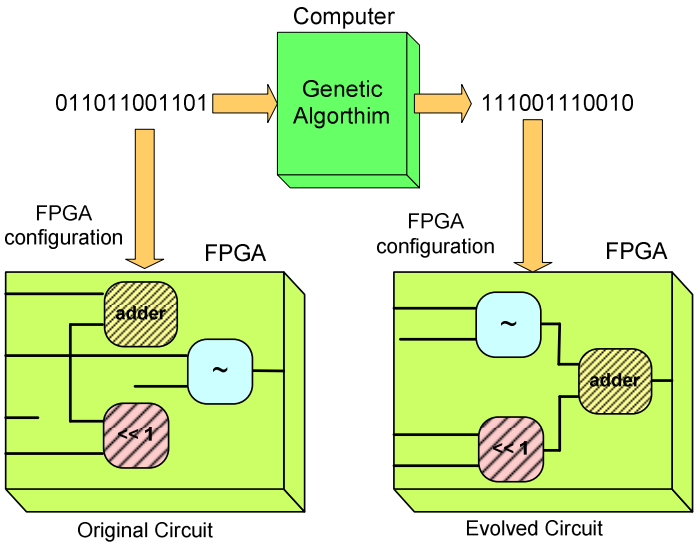


Figure 3-21. Schematic representation of functional level evolution.

3.5.4 Cartesian-Based Virtual FPGA Architecture

This type of architecture is based on Cartesian genetic programming which has previously been described. The circuit architecture consists of a two dimensional array of nodes linked via a Cartesian co-ordinate system, with the data flow moving from left to right. The circuit can therefore be expressed by a configuration bit stream that describes the connections between the nodes, and the functions of the nodes. To overcome destructive architectures, only feedforward connections between nodes are allowed. In general, Cartesian genetic programming cells would have multiple inputs and outputs, and feedback would be allowed.

A Cartesian based architecture is shown in Figure 3-22. The inputs and outputs of the FPGA, as well as the outputs of the nodes and the nodes themselves are numbered. The function of the node is linked to the number (in this example, 11 is an AND gate whereas 12 is OR gate). The chromosome is a one dimensional array which represents both the routing and node function for the complete circuit. In the example circuit provided, each node and input connections are represented by four numbers. The first three numbers describe how the node inputs are connected, and the fourth number describes the function of the node. This sequence is repeated in the array until the full virtual FPGA is described. Note all connections are feedforward, as no feedback connections are allowed. After reproduction, any damaged chromosome will be required to be repaired.

In this example, the first AND gate is implemented as a two-input gate connected to A and C (note the third input is always held high). The first OR gate is implemented as a two input gate (note the third input is always held low) connected to the outputs of the preceding two AND gates. In this way the one dimensional array can describe the connections of the circuit, and it is this array that can have genetic algorithms performed on it.

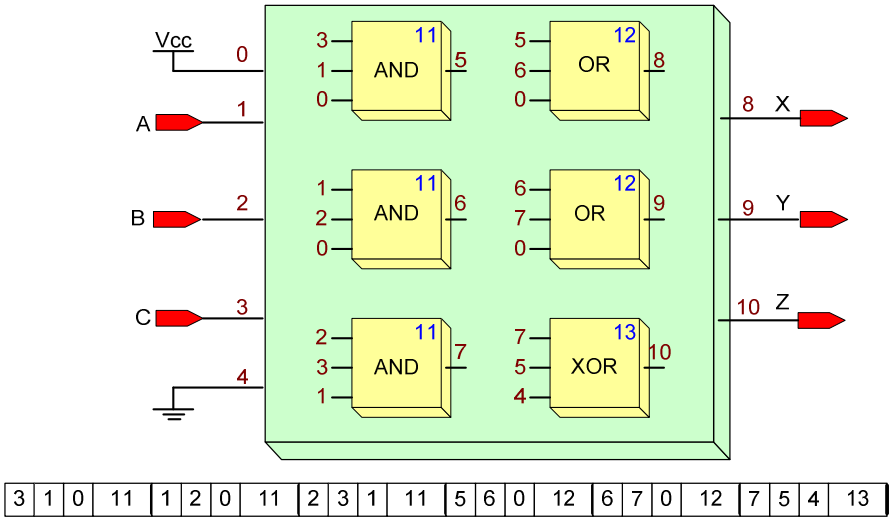


Figure 3-22. Cartesian architecture showing the functional elements numbers and the string that describes their interconnections.

Several examples of virtual FPGA based on the Cartesian programming architecture have been used to evolve digital circuits. Higuchi et al. [85] used this approach to generate two high level applications, an adaptive equaliser and a lossy data compression. The researcher used a two dimensional array of 'Programmable Function Units' as shown in Figure 3-23. The programmable function units were capable of adding, subtracting, if-then, sine generator, cosine generator, multiplying or dividing. These programmable function units and the routing between them were controlled by the configuration bit stream. The unit was comprised of one hundred nodes or programmable function units in an array of twenty columns by five rows. Each column could access the data from the previous column, or directly from the inputs. There were two inputs with the output fed back to the input. The data was a floating point number.

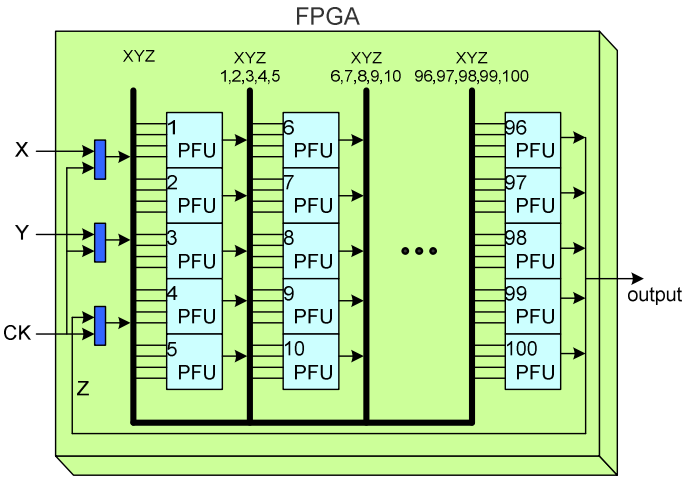


Figure 3-23. Cartesian architecture showing programmable functional units for signal processing.

Sekanina [86] used a virtual FPGA based on Cartesian programming for image processing. In his original studies he found evolving at the gate level virtually impossible for complex circuits due to the large search space, whereas good results could be produced by evolving higher level configurable functional blocks. In his research, Sekanina created a configurable function block that had two eight-bit inputs and one eight-bit output. The functions within this logic block are shown in Figure 3-24.

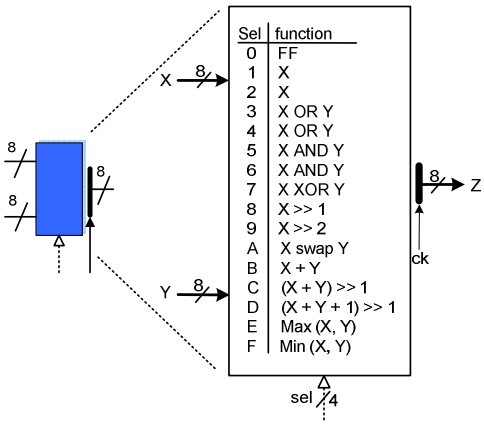


Figure 3-24. Functional listing of the lookup table for a configurable function block.

The configurable function blocks were laid out in a two dimensional grid of seven columns by four rows, with a final block on the output, as shown in Figure 3-25.

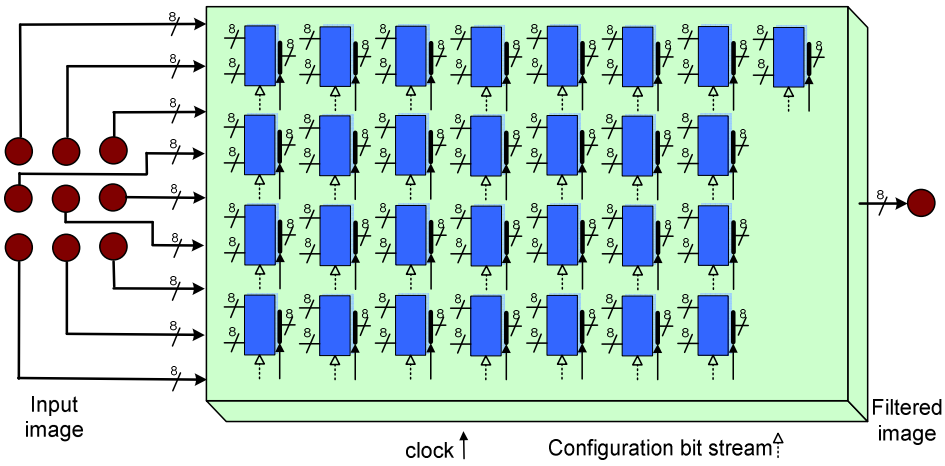


Figure 3-25. Cartesian architecture showing configurable function blocks for image processing.

The configuration bit stream controls the routing between the configurable function blocks as well as the function within the block. In order to reduce the routing overheads, connections between blocks were limited. The circuit was evolved to minimize the difference between a corrupted image and the uncorrupted original. In comparison to conventional filters, it was found that the evolved circuit produced a better quality picture and in some cases, more efficient use of resources.

Sekanina [87] has also investigated evolvable intellectual property cores. He proposed that these cores should be able to be reused in a similar fashion to standard intellectual property cores, and that they can perform autonomous evolution of their internal circuits. The cores will be made available as hardware descriptive language modules, comprised of a virtual reconfigurable circuit and a genetic unit controller which can be synthesised into any reconfigurable device. The evolvable intellectual property will be stored in a standard component library and downloaded to the FPGA, however, when running they will evolve their circuit autonomously. These cores can be reused in a similar way to intellectual property, for example an evolvable digital filter. The genetic controller will perform genetic operators, such as crossover, mutation, and reproduction, and will have memory storage for fitness, although it does not perform fitness testing as this is a specific task. Note that the evolutionary intellectual property does not have to reside in a partial reconfigurable FPGA.

Moreno Arostegui et al. [88] suggested a FPGA architecture that simplifies the routing requirements inside the FPGA. Currently, FPGAs are designed with a high degree of flexibility which requires a complex routing system. This routing system demands a large search space which makes it difficult to evolve a complex system. They proposed

a hierarchical layered organization of a regular two-dimensional array of cells whose routing strategies are part of the hierarchical layer allowing incremental routing paths amongst the functional cells. This allows the addition of more functional units without having to calculate complex routing strategies and is better suited towards evolvable hardware.

Wang et al. [89] used a Cartesian based virtual FPGA with a hardware genetic algorithm to evolve a character recognition FPGA circuit. The virtual FPGA had 30 inputs which were connected to the 5x6 pixel array. (Note in their experiment they used a lookup table with the character pixels rather than an actual camera.) These pixels were then passed through a four layer functional block array as shown in Figure 3-26. The blocks had simple logic functions, and selection. The virtual FPGA had sixteen outputs that were associated with a character ranging from A to P.

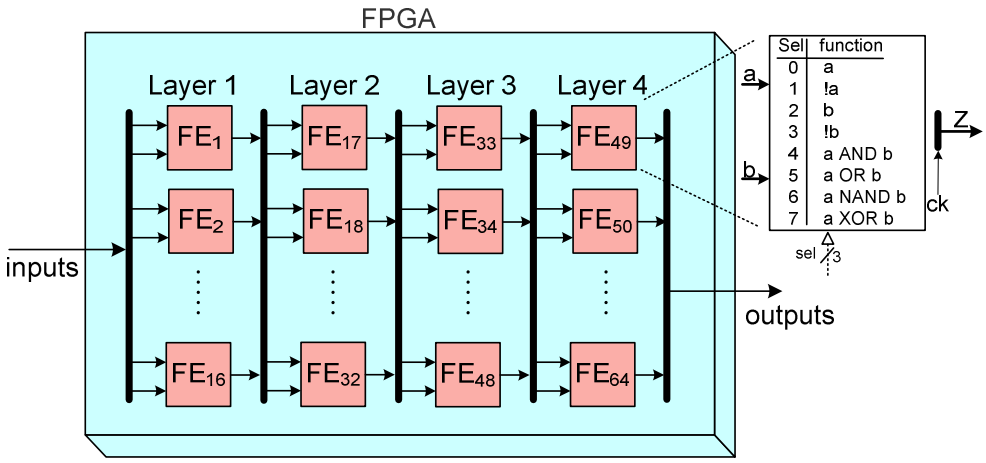


Figure 3-26. Virtual FPGA architecture used for character recognition.

3.6 Chromosome Length Reduction

The most task orientated process is fitness evaluation; however, this is linked to the size of the genetic algorithm and the search space that needs to be explored. If the chromosome length can be reduced, then the search space and the corresponding time taken to evolve is shortened. The difficulty with a standard genetic algorithm is that as the complexity of the problem increases, so too does the possible permutations and the time required to evolve.

The chromosome used to describe a circuit can be large and impractical to evolve. Proposals for using a virtual FPGA with limited routing and function level blocks in order to reduce the chromosome length have been previously discussed. Alternative

methods are to alter the one to one mapping between the genotype and the phenotype and thus the length of the chromosome. In biological terms the genotype is the gene inside the chromosome and the phenotype is the expression of that gene. For example, a gene determines eye colour, whereas the phenotype is the actual eye colour the gene produces. For example, blue or green or brown is the phenotype or expression of that gene.

The genotype-phenotype map relates to how a genotype will be expressed as a physical trait. The genotype represents the exact genetic makeup, for example, the bit or number sequence that is being manipulated by the genetic algorithm. The phenotype represents the actual physical properties, for example, what type of gate or function is expressed. The way that the genotype affects the phenotype is called the genotype-phenotype map. Evolvable hardware could be made more efficient if the genotype-phenotype mapping were made more complex, allowing the genotype to be reduced in size. A good example of a genotype-phenotype map is to imagine you are building a paper plane using a list of instructions on where to fold paper (A – Z). The genotype could be GABKJAAND; the phenotype would be the expression of the genotype, which is the shape of the paper plane. Reducing the genotype moves the complexity to the genotype-phenotype mapping.

3.6.1 L-system mapping

The L-system was originally used in biology to predict how simple multi-cellular systems will grow. It is a mathematical system that uses a rule-based system to show how the structure would grow from a particular starting point. For instance, if the rules were X changes to a Y, Y changes to X, Y then the following pattern would occur as shown in Figure 3-27.

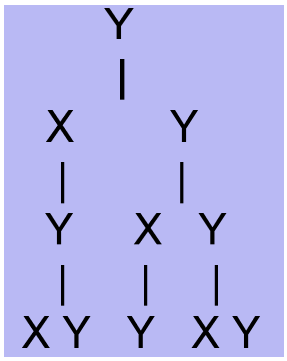


Figure 3-27. An example of a growing structure based on L-system mapping.

In genetic programming, the set of rules is the genotype and what they produce is the phenotype. To use L-system mapping with evolution, the rules themselves are the chromosome and these are evolved. Thus, we evolve a population of rules, where the chromosome stays fixed while the phenotype can vary in length depending on what the rule produces.

Haddow et al. [90] used this method to implement a routing structure in a grid array of S-blocks. He had two modes, change, (where the chromosome stayed the same size but changed the S-block operation), and growth (where the chromosome increased in size and included more S-blocks). Thus, the chromosome would increase in size as the complexity of the circuit increased. Note this system does not have a one to one mapping between the genotype and the phenotype, therefore it has less search space.

Schaefer [91] used genetic programming using the L-system for path planning in robots. An example of an evolved rule was A → < f b - s f +> where f is forward one step, b is back one step, s is stop, + is turn right, and – is turn left. He used this chromosome to evolve a controller whose objective was to drive the robot towards a light whilst avoiding obstacles.

3.6.2 Variable Length Genetic Algorithms

A normal genetic algorithm uses a chromosome with a fixed length that describes the solution to the problem. A variable length genetic algorithm is similar to the normal genetic algorithm, except that it is able to increase or decrease the length of its chromosome. This makes it capable of increasing its complexity as it evolves for more difficult tasks.

Kajitani et al. [92] used a variable length genetic algorithm for hardware evolution by initially reducing the chromosome, by not incorporating all possible permutations of the programmable logic device switching elements. As the circuit evolved, the chromosome was increased to include more switching elements to allow more complexity in the circuits.

Iwata et al. [93] also used a variable length genetic algorithm to evolve a programmable logic device for pattern recognition. The chromosome consisted of the location and connection types of the fuse array that configures the programmable logic device. Initially, the chromosome was small, limiting the number of input connections that the

pattern recognition could use. However this chromosome could be increased in size, connecting more resources as required. It was found that the average evolved variable length genetic algorithm chromosome had a chromosome length of 187.6 bits as compared to a standard genetic algorithm chromosome length of 840 bits.

3.6.3 Species Adaptation Genetic Algorithms

Thompson et al. [94] used variable length chromosomes in his use of species adaptation genetic algorithms (SAGA). Standard genetic algorithms are not suitable for cognitive structures as these require a slow rather than abrupt change. Robotic controllers need to be able to move from simple to more complex architectures which suit the use of a variable length genetic algorithm. Thompson et al. investigated the use of species adaptation where similar individuals are likened to a species. He used a method of hill crawling, balancing the need for exploitation (maximizing the fitness locally) with exploration (finding new fitness maxima). Thompson et al. considered tournament as the best method of selection because of its ability to adapt to high mutation rates while maintaining hill crawling features.

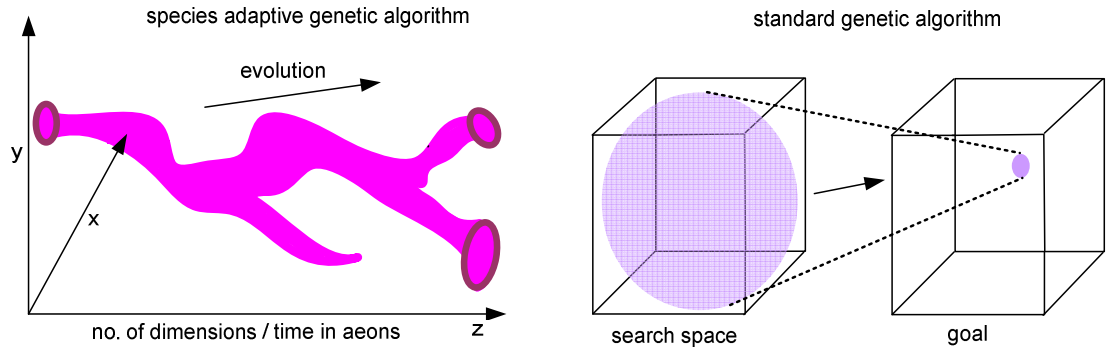


Figure 3-28. Diagrammatic comparison of the search space for SAGA and a standard genetic algorithm.

A standard genetic algorithm chromosome will start with a random distribution within the search space and narrow down to a high fitness individual (goal) as the evolution progresses, as shown in Figure 3-28. In comparison, SAGA has a chromosome that is capable of increasing in length, and is associated with the evolution of a species or homogenic groups of individuals within the population. This gives the possibility of the chromosome splitting into separate species, or extinction of species.

3.6.4 Compact Genetic Algorithms

Gallagher et al. [95] investigated using compact genetic algorithms for evolvable hardware to reduce the size of the chromosome and subsequent decrease in the search space. Rather than having a population of numerous chromosomes, the compact genetic algorithm has only one chromosome. Each parameter in the chromosome has a probability ranging from one to zero. This chromosome is used to generate offspring where the probability parameter for each bit is used to determine if the associated bit in the offspring will be a one or zero. At the start of the evolution, all the probabilities within the chromosome are set to 50%, thus the offspring will have a random bit pattern. After each generation, the bit pattern of the better individuals will alter the probabilities in the parent chromosome. As the evolution progresses, these probabilities move towards 0% or 100%.

3.6.5 Morphogenetic Algorithms

In nature a separation between the genotype and phenotype allows complex organisms to evolve. The genotype (bit pattern) and the phenotype (generated circuit) is separate, with the phenotype being generated from the genotype. If the genome expressed is a growth process (morphogenesis), rather than an explicit configuration of a circuit, then complex forms can be created from a simplified genome. Natural evolution evolves simple structures which go on to evolve into more complex systems. Biology maps genotype to phenotype through regulated gene expression. Mapping can be performed by using a set of production or grammar rules, thus the evolutionary process can work on the grammar rather than that of a program modifying system. A common production rule is the Lindenmayer system (L-system) which was originally used to model the growth in plants. The parallel nature of this system suits evolvable hardware.

Lee and Sittle [96] used a cell based morphogenetic model for hardware evolution. The mapping of the genome was via the Xilinx JBits application programming interface. The cell structure was closely based on the Xilinx Virtex architecture, with each cell stored within a configurable logic block slice. These researchers created and packaged a chromosome that represented the chromosomes from nature, using a variable length chromosome with base-4 encoding. This chromosome imitated the structure of a biological cell.

Gordon and Bentley [97] modelled a circuit based on a cellular structure where the chromosome and proteins control the function of the cell. The proteins were binary state variables that were present, not present or don't care. The chromosome was a set of rules based on the L-system which would affect the cell and its proteins, either creating more proteins or altering the function of the cell. The cell was modelled as a functional unit which incorporated input/output, protein detector/generator and a function generator.

3.6.6 Incremental Learning

Incremental learning, or increased complexity evolution, is used to overcome the problem of long configuration bit streams. Evolution occurs discretely on small units such as logic gates and functional blocks. These evolved blocks are then used in a second stage evolutionary process to create more complex circuits or systems. This design can be likened to a bottom up process. The fitness function can either be a subset of the complete fitness function, or it can be designed for individual tasks which are combined to create a global fitness function.

Torresen [98, 99] used incremental learning to create a road image recognition system. The image system had several processing stages including noise removal, thinning, and recognition. The information was to be used for autonomous driving using the road markers as a reference. This system was too complex for a one step evolution. Torresen based the architecture on an array of logic gates, using the Xilinx 6200 FPGA. The inputs to this architecture was a 4x8 pixel image, and the output was turn right, turn left, move straight ahead. A comparison was made between evolving the system directly, and using subsystems, with each subsystem having a limited amount of outputs. There was a substantial reduction in the number of generations required for a successful evolution when incremental learning using subsystems was used.

3.7 Hardware-Based Genetic Algorithms

An important feature of FPGAs is their parallel nature. The hardware descriptive language produces digital circuits that execute simultaneously, rather than a computer program that executes instructions in a sequence. This technique can be implemented in evolvable hardware to speed up the evolutionary process. The hardware genetic algorithm can be split into three functions: reproduction, selection and fitness

assessment. These can be performed in either hardware, software or a mixture of the two.

3.7.1 Mutation Only Hardware Genetic Algorithms

The genetic unit can comprise either a mutation unit alone, or a mutation unit with crossover. There has been research into using these two operators independently or together. When used together, the crossover is used to select the best trait within the population, and mutation is used to provide diversity as the population converges on local maxima. However, research has also investigated mutation only genetic algorithms which do not use crossover. This has been described in chapter two.

Wang et al. [89] created a hardware genetic algorithm as shown in Figure 3-29 that had five basic systems. These are: a) a random number generator; b) population memory; c) best chromosome; d) a mutation block; and e) a mutation rate selector. The mutation rate was part of the chromosome and was able to be varied each time a new individual was created. The hardware genetic algorithm had a population memory of four, which were evaluated and the best one was then used to generate a further four offspring. The hardware genetic algorithm used mutation only as the reproductive operator. The mutation rate varied between 0.2 to 1.6 %. This hardware genetic algorithm was used in conjunction with a virtual FPGA to implement a character recognition circuit.

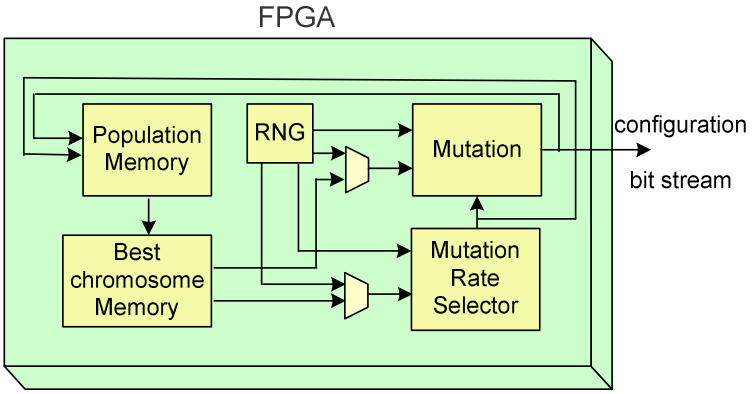


Figure 3-29. System blocks and their interconnections for a mutation only hardware genetic algorithm.

Sekanina et al. [57, 58] created a hardware genetic algorithm using mutation only as the genetic operator as shown in Figure 3-30. The virtual FPGA used functional blocks grouped in a Cartesian genetic programming array. A population of individuals was generated using a random number generator, and mutation was performed on the

reproduction of the new off-spring. A hardware fitness unit was also used which compared the outputs of the virtual FPGA with the desired or wanted outputs. The fitness was the sum of the correct outputs relative to the inputs. Simple circuits such as multiplexers, adders and parity encoders were generated.

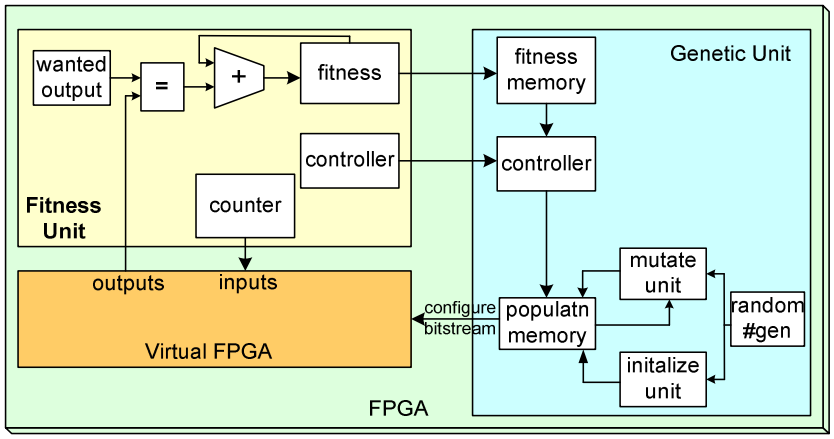


Figure 3-30. System interconnections of a virtual FPGA, a mutation only genetic algorithm and fitness evaluation.

3.7.2 Crossover and Mutation

Shackleford et al. [100] created a hardware genetic algorithm using both crossover and mutation operators. Two parents were loaded into the crossover system, with each bit of the parent connected to two-bit multiplexers as shown in Figure 3-31. A crossover template was used to select which point in the chromosome would be cut, depending on the input pattern of the crossover template shift register. This allowed a range of cut points ranging from single point, to multipoint, to crossover uniform. The crossover pattern was generated by comparing a randomly generated number to a threshold value, producing a one or zero which is serially fed into the crossover template. Increasing the threshold value increased the number of cut points.

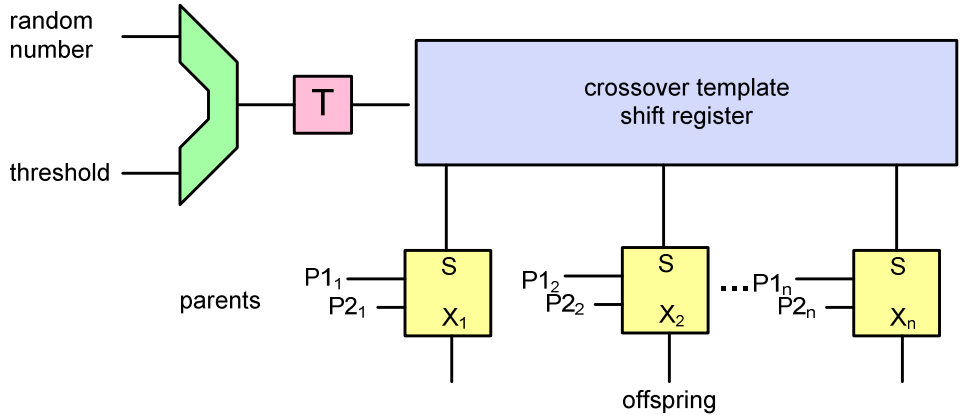


Figure 3-31. Hardware crossover using a crossover template two bit multiplexers.

Shackleford et al. also used a template for the mutation operator as shown in Figure 3-32. A mutation will occur when two 'ones' occur simultaneously in the shift register bit positions giving a logic one after the AND gate, and inverting the bit after the Exclusive OR. To increase the efficiency of the hardware they used a steady state genetic algorithm model rather than a generational genetic algorithm.

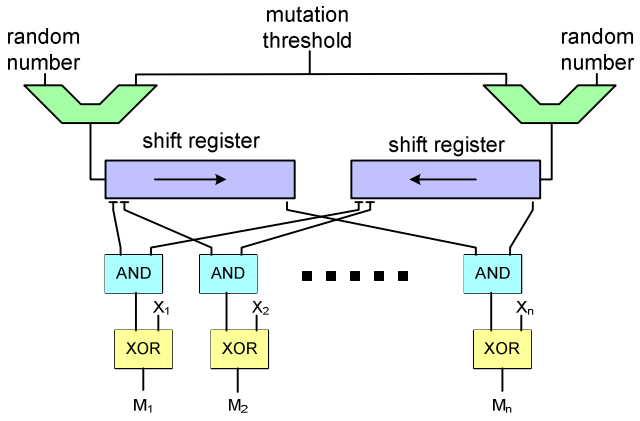


Figure 3-32. Hardware mutation using shift registers.

3.7.3 Pipeline Processing

Maruyama et al. [101] used the pipelining and parallelism features of a FPGA to speed up the evolutionary process and applied this to solve the classic knapsack problem. They used two FPGAs, one for the hardware genetic algorithm including crossover, selection and mutation, and the other for the fitness testing. As shown in Figure 3-33, the individuals are serially clocked into the genetic algorithm. The blocks inside the genetic algorithm are the randomization block used to change the order of the population, and the selection block which uses tournament selection between two adjacent individuals. After passing through these two blocks, the population is then

randomized again. The crossover block is performed with counters and switches, with the counters incrementing to a random number and then flipping the switch between the two individuals being crossed over. Finally the chromosome passes through the mutation block which randomly inverts a bit. The resulting offspring was sent to the second FPGA for fitness evaluation. The researchers incorporated several of these pipelined genetic algorithms using the island model selection process.

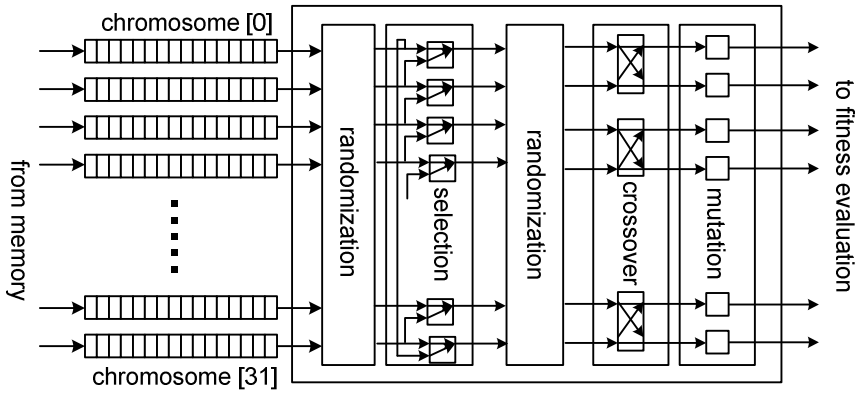


Figure 3-33. A pipelined hardware genetic algorithm with both crossover and mutation.

Yang et al. [102] used a hardware genetic algorithm as shown in Figure 3-34, to evolve a functional digital circuit that would implement high performance digital image filters. They used a virtual FPGA modelled on a Cartesian-based array of functional logic blocks. The genetic algorithm was created in hardware with mutation only; it used elitism for its selection operator and had a fixed population of sixteen individuals. Only one individual was kept after each generation, and this was mutated to provide fifteen offspring to replace the population. The operation was as follows: a) the image was passed to the input buffer and then to the virtual FPGA; b) the virtual FPGA was configured by the individual sent from the internal memory and its fitness accessed; c) the selection process determined which individual had the best fitness and notified the interface memory; d) the best individual was kept and sent to the mutation unit to generate fifteen more individuals; then e) the process was repeated until the required fitness was reached. Adaptive mutation was implemented with the mutation rate being inversely proportional to the fitness. The fitness was derived from the mean difference per pixel, which was the difference between the original and filtered image. The results of this study found that the hardware implemented genetic algorithm was faster than a similar software algorithm by two orders of magnitude.

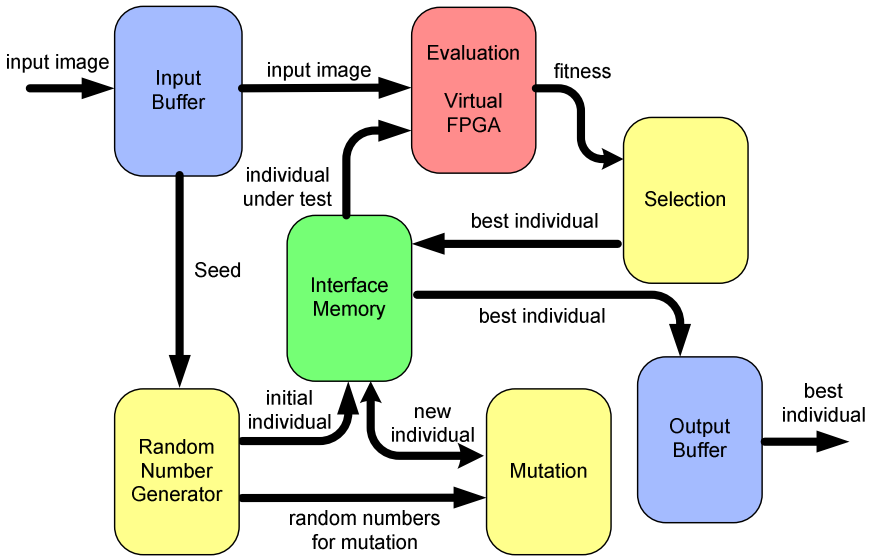


Figure 3-34. Block diagram of a mutation only hardware genetic algorithm.

3.8 Examples of Robotic Controllers

There are several examples of evolvable hardware being used to control a robot using either the Xilinx 6200 series of non-destructive FPGAs, or virtual FPGAs.

As previously discussed, Krohling et al. [64, 103] used evolvable hardware to control the motion of a Khepera robot for obstacle avoidance. In a similar fashion, Tan et al. [104] used evolvable hardware with the Khepera robot to successfully follow a light source whilst avoiding obstacles in real time. Tan et al. also investigated how the effects of a traction fault were overcome by the evolutionary hardware, as a fault like this could not be taken into account with normal software. These researchers used a turret that contained a Xilinx 6216 FPGA which could be attached to the Khepera robot. The infrared sensors from the robot were sent to the FPGA, and the output from the FPGA was sent to the robot's motors. Intrinsic evolution using the Khepera robot's internal Motorola 68331 microprocessor was performed to evolve the Xilinx FPGA configuration bit stream. The fitness was determined by the response of the robot to the light source.

Okura et al. [105] evolved a hardware controller for a Khepera robot for obstacle avoidance using the Xilinx XC6216 turret. The FPGA was configured by the Khepera robot microcontroller using C functions. These functions could initialise and disable the FPGAs ports, set communication between FPGA and microprocessor, and configure the cells within the FPGA. This last function was important as each cell could be specifically altered allowing a reduction in the chromosome length, as redundant bits

were not included. The fitness was determined from the distance travelled before an obstacle was hit, as well as how many paths the robot took, (preventing a simple forward backward motion from having a high fitness). Okura et al. also compared a microbial and standard genetic algorithm finding that while both algorithms could successfully evolve controllers, the microbial could outperform the standard.

Seok et al. [67] used genetic programming to evolve a robot controller that could move towards a light whilst avoiding obstacles. An example of the tree structure based on the robot path for this problem is shown in Figure 3-35, where the desired path is to avoid obstacles while continuing to move towards a light. If an obstacle is found, it will move down the sub-tree depending on the direction of the light source and obstacles until reaching the bottom layer nodes which specify the motor action. The direct implementation of this tree in hardware is difficult as it is inefficient in both hardware and routing resources, and the crossover operator is difficult to perform. To overcome these problems, Seok et al. created a linear representation of the tree which expressed as binary strings with each path illustrated from the top node to a bottom node.

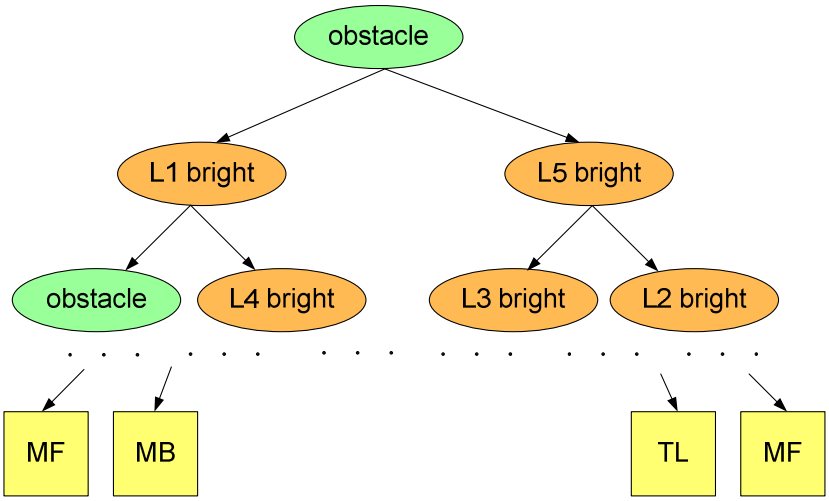


Figure 3-35. Example of the tree structure required for obstacle avoidance and light following.

Lund et al. [106] investigated evolving the body of the robot as well as evolving the robot controller. The controller was simulated on a specific hardware configuration where the hardware was able to include the circuit on which the controller was implemented, as well as at a more advanced level, the motors, sensors and physical structure of the robot. For example, a small wheel might have been useful for a fast turning robot whereas a large wheel was more useful for a slow turning robot. To evolve the robot, Lund et al. used part of the chromosome to determine its body shape as well as the robot controller. This chromosome was used to configure the robot's body

and its actions in simulation. The researchers used Lego parts for the evolution so that the robot could be easily built. They then went on to develop ears for a Khepera robot which could follow a sound source much like a cricket would when finding a mate.

This chapter has summarised the concepts of using a genetic algorithm to evolve hardware. It has shown three main processes, genetic compilers, genetic programming, and virtual FPGAs. In particular, the use of the virtual FPGA and its application in robotic control has been examined.

Chapter 4

Chapter 4: A Review of Artificial Neural Networks and Fuzzy Logic Controllers and their use in Evolutionary Robotics

The previous chapters have presented the current research in evolving robotic controllers using both software programs and hardware circuits. Although not specifically part of the research presented in this thesis, artificial neural network and fuzzy logic robot controllers have been widely used in the field of evolutionary robotics. This chapter identifies and examines the extensive research that has been performed in the field of evolving artificial neural networks and fuzzy logic robotic controllers.

4.1 Evolution of Artificial Neural Networks Robotic Controllers

4.1.1 Artificial Neural Network Overview

An artificial neural network is a network of interconnected neurons that are modelled on neural networks found in nature. Each neuron has two or more inputs whose values are modified by a weighting factor. These weighted inputs are then fed into a summing input. The output of the neuron will fire when the sum of the inputs exceeds a threshold value. A model of an artificial neural network is shown in Figure 4-1. The individual inputs are each multiplied by a weighting factor, (ω_{km}). This is then passed to a summing network which has an input bias that is used to adjust the sensitivity of the neural network. When the summing input reaches a predetermined threshold level, the activation unit will output a value whose shape is determined by the activation function $\Phi()$.

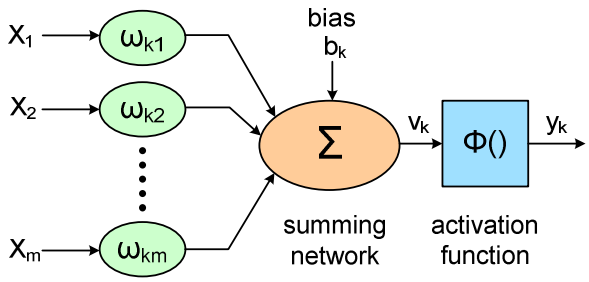


Figure 4-1. Diagrammatic representation of an artificial neuron.

The artificial neural network can be described by mathematical parameters such as the neuron activation value, the decay rate, the bias term, the firing rate, and the strength of the synaptic connections from neuron to neuron. The mathematical model of the above neuron is shown in Equation 4-1.

$$y_k = \varphi\left(\sum_{j=0}^{m} w_{kj} x_j + b_k\right) \qquad \textbf{Equation 4-1}$$

Where

x is the input,

w is the weighting,

b is the bias,

φ is the activation function,

The activation function sets the type of output of the artificial neural network. There are many different types including step, sign, linear, sigmoid and tanh as illustrated in Figure 4-2. The most common output is the sigmoid function.

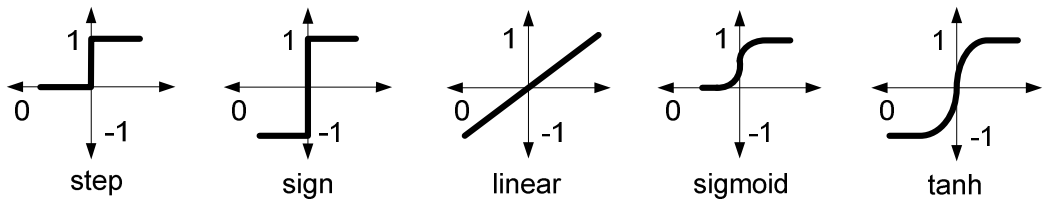


Figure 4-2. A graphical representation of activation outputs from an artificial neural networks.

An artificial neural network is comprised of layers of neurons where each neuron in one layer is connected to every neuron on the following layer. The simplest network is the single layer where the inputs connect to one layer and these then connect to the outputs as shown in Figure 4-3. More complex artificial neural networks will have layers in between the input layer and output layer, called hidden layers, which do not connect to the external world.

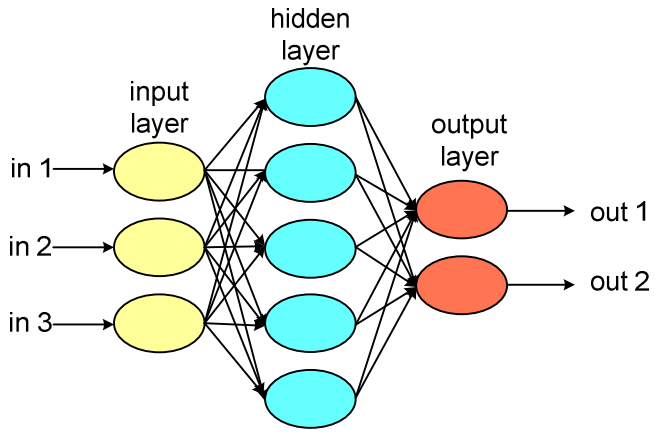


Figure 4-3. A single layer artificial neural network.

The aim of the artificial neural network is to provide a correct response on its outputs depending on its inputs. This in turn is dependent on the weightings of the interconnections to each neuron. Before the network can operate, it must first be trained by going through a learning phase. In this phase the weight associated with each neuron interconnection is adjusted so that the network can provide the appropriate response to its inputs. There are three types of learning including supervised, unsupervised, and reinforcement.

- **Supervised learning:** This is where the output pattern of the network is known for each example of input patterns. The input pattern is presented to the network and the output recorded. The weightings on the neuron interconnections are then adjusted to set the output of the network to best match the desired output. The difference between the required output and the actual output is called the error. The artificial neural network weights are adjusted to minimize this error, thus the neurons are trained to either fire or not fire for different input patterns.
- **Unsupervised learning:** The weight of each interconnection is adjusted on the basis of the input pattern alone. The network learns by adjusting its weights so that similar inputs cause similar outputs. This forms the input pattern into a number of meaningful classes.
- **Reinforcement learning:** This is a combination of supervised and unsupervised learning. Although no desired outputs for a given pattern of inputs is provided, the network is told if it is learning in the right direction.

The learning process can occur either offline, where the neural network is non operational while it learns, or online, where the neural network learns while it operates. Normally supervised learning is offline and unsupervised learning is online.

When operating the artificial neural network and the input pattern is presented, the neuron will respond as trained. However if the pattern is not present, then the neuron will fire depending on the firing rule. An example of a simple firing rule would be to use the hamming distance, which is a measure of how far the input pattern differs from the trained pattern.

4.1.2 Types of Artificial Neural Networks

Feed Forward network: The artificial neural network information moves in one direction only, starting from the input layer, moving forward to the hidden layers and then to the output layer.

Recurrent network: The artificial neural network information flows in both directions, making it possible for outputs of one layer to feed back to inputs of previous layers. This artificial neural network is time dependent as the feedback paths contain the information from the previous state, allowing a type of pseudo memory. This gives the artificial neural network the ability to perform sequence prediction tasks. A fully recurrent network is not layered, as every neuron connects to every other neuron in the network.

Spiking network: The artificial neural network will replicate typical brain activity which sends inter-neuron messages in brief spikes of short duration, rather than as a continuous signal.

Weightless network: The artificial neural network uses only binary values on its inputs and outputs with no weights on the neuron interconnections. The neuron function is stored in a RAM lookup table. Learning consists of changing the contents of the lookup table parameters. The advantages of this type of network are that they can learn in one shot, or epoch, and they do not require multipliers. Its limitation is that if the number of inputs is high, then a large amount of memory is required for each neuron.

4.1.3 Creating Artificial Neural Networks on a FPAA and FPGA

Creating an artificial neural network on a FPAA

Artificial neural networks created for software program systems are comprised of a series of additions and multiplications which are calculated sequentially. These same artificial neural networks can be created on a FPAA. The advantage of implementing an

artificial neural network on a FPAA is that adders and multipliers can be easily implemented and that computations can take place in parallel and asynchronously, limited only by propagation delays in the circuit. The disadvantage is that the number of neurons is limited to the resources within the FPAA device.

Creating an artificial neural network on a FPGA

Artificial neural networks have also been developed on FPGAs, although it is more difficult than using a FGAA as the extensive use of floating point arithmetic multipliers and the nonlinear activation function of the neurons is resource intensive for FPGA applications. Several methods have been developed to overcome the use of floating point multiplier in FPGAs. A description of some of these is provided below.

Pulse Stream Arithmetic: The floating point number can be encoded as a pulse stream rather than an integer number. For example, a fractional value such as 7/16 can be described as 7 pulses in a 16 bit window. The inputs to the neurons are a pulse stream which can be gated. The addition process can be performed by simply ORing the separate lines going into the neuron, while multiplication is performed by ANDing these input lines. Note the signals are derived by synchronous non overlapping clock pulses. These pulses are then passed through an up/down counter to produce a binary step function on the output of the neuron. An artificial neural network of this type was implemented by Lysaght et al. [107].

Power of two arithmetic operations: In order to avoid multipliers, the weighted values can be limited to powers of two, or sums of the power of two. Thus addition and multiplication can be performed with the use of shift registers which are easier to implement in an FPGA. Marchesi [108] implemented this technique to create a neural network that was used in pattern recognition. The neural network architecture was based on back-propagation, while the learning phase required real arithmetic and was performed offline. Once the weights were learnt, the corresponding powers of two were loaded into the network.

Conversion of real numbers to integers: The reduced complexity of the integer multiplier requires fewer resources than a floating point multiplier in a digital circuit. However the use of integers leads to a loss of precision in the final design.

Fixed floating point numbers: The binary number is broken into two separate parts: one for the integer and the second for the fraction. The binary number has less range than a floating point number. Prieto et al. [109] used a 16 bit sign magnitude binary format with 6 bits to represent the integer and sign and 10 bits for the fraction. They created a three layer network and compared this binary network with a decimal network with successful results.

Stochastic arithmetic: In a similar manner to a pulse stream, stochastic arithmetic uses values that are represented as a pulse density, where the numeric value is proportional to the density of the 1's in a bit stream. This allows addition and multiplication to be performed with simple digital architecture. Multiplication can be performed by a simple AND gate, and addition by the use of an OR gate. Bade and Hutchings [110] used this approach in their implementation of an artificial neural network on a FPGA.

Weightless neural network: This network has been described previously in this chapter. The advantage of this network is that it does not require multipliers, making it more efficient for FPGA implementation. Hannan Bin Azhar and Dimond [111] developed a RAM based weightless neural network on a FPGA for a robot controller which was used for obstacle avoidance.

4.1.4 Evolving Artificial Neural Networks

As an artificial neural network initially goes through a learning cycle (of either supervised, unsupervised, or reinforced learning to modify the weights of the artificial neural network), it is trained to react to its environment in a particular way. Evolution can also be used to create an artificial neural network by using a genetic algorithm to evolve either the weight functions of the neurons, the neural network structure, or the learning rules. Examples of evolved networks for robotic controllers in both software and hardware (FPAA and FPGA) are described in the next section.

Evolving artificial neural networks in software for robotic controllers

Bianco and Nolfi [65] evolved a neural controller to control a simulated two-finger robotic arm. The aim of the arm was to grasp objects of different shapes based on the tactile information coming from the hand sensors, while also dealing with the constraints of gravity and collisions. They used a two layer network with fifteen neurons on the input sensors and nine neurons for the motor control. Six of the inputs

were contact sensors and nine were proprioceptive sensors which showed the angular position of the arm and fingers. The outputs went to nine motors controlling the actuators for each joint. The chromosome held the connection weights and biases of each neuron in the controller. The population size was 100, with a two percent mutation rate. Fitness was initially set by the number of times the object was grasped, and then by the number of objects grasped within a certain time frame.

Mondada and Floreano [112] evolved an artificial neural network for a Khepera robot for three separate tasks including navigation while avoiding obstacles, homing, and gripping. The artificial neural network was a multilayer perceptron with recurrent connections on the hidden layers (except for the navigation task which had no hidden layers). Sensors were connected to fixed inputs of the neural network and its outputs were connected to the motors as shown in Figure 4-4. The genetic algorithm used linear scaling for its selection, with single point crossover and creeping mutation for its reproduction. The population size was one hundred, with the chromosome showing the neurons synaptic weights and thresholds encoded as floating point numbers.

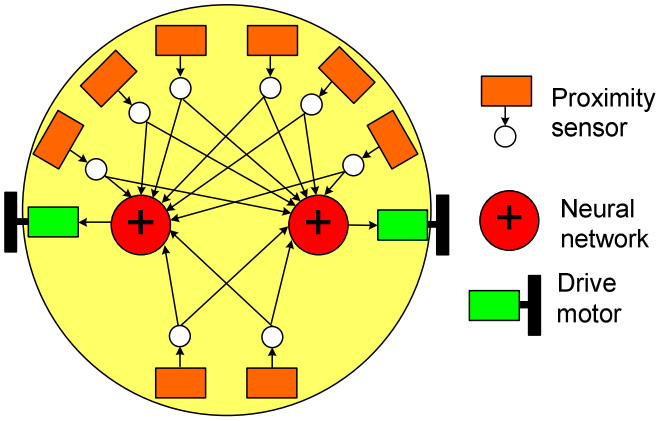


Figure 4-4. The neural network configured for navigation on a Khepera robot.

Nelson et al. [113] evolved artificial neural networks that were used for robotic controllers to play a game called ‘Catch the Flag’. The evolved robotic controller required 150 inputs to process its video sensor, and produced two drive wheel command outputs. Several network structures were used including feed forward, and feed back topologies, with a range of activation functions for each neuron. The chromosome was a combination of two arrays with the first containing the connectivity and weighting relationships of the network, and the second containing the neuron type. Thus the chromosome could describe varying sized networks and their connections. The evolution of the artificial neural network was in simulation. The fitness was calculated

with two weightings, the first selected for minimal competence to successfully complete the task, and the second based on whether they won or lost.

Kim and Cho [114] evolved artificial neural networks based on cellular automata to create a robot controller for a simulated Khepera robot using incremental evolution. Cellular automata were used to create the artificial neural network in 'CAM-Brain', a system based on artificial intelligence that was first developed in Japan. The network grew to the cellular automata rules. The genetic algorithm was applied to the cellular automata rules thus altering the structure of the neural network. Initially basic behaviours were evolved such as move straight, avoid obstacles, and follow a light. Maes action selection mechanism was then used to determine which evolved basic behaviour to apply.

Tuci et al. [115] evolved a simulated robotic controller based on the Khepera robot to move towards a light source. The evolution was performed on a recurrent artificial neural network that could be configured to any size and connectivity. The network had ten inputs for the sensors and four outputs for the motors. The researchers employed the novel technique of keeping the weights of the network constant and modifying the activation parameters of the neuron. The chromosome detailed the input and output connections of the neuron and its associated activation threshold and decay values. The genetic algorithm used elitist selection with a population size of 200. Micro-mutation was used to randomly alter parameters within the chromosome, while macro-mutation was used to operate on a complete chromosome (that was to add or delete a complete chromosome, thereby adding or deleting a neuron).

Leon et al. [116] looked at a discrete-time recurrent artificial neural network with two variants, plastic artificial neural network and feed forward artificial neural network, to see how they could be modified by a genetic algorithm for robotic navigation. They used two classes, recurrent and non-recurrent. The development of an evolved controller which could navigate towards a light was successfully achieved by evolving the weights in the artificial neural network.

Abhishek et al. [117] evolved a robot controller for the Khepera robot using a recurrent neural network configured by a variable length chromosome. The researchers used a recurrent neural network as it was capable of memory and non reactive behaviour. The chromosome described the whole network including the number of neurons, the number

of inputs and outputs, and the connections, weights, threshold and delay times of each neuron. The number of layers in the network was kept constant; however the number of neurons in each layer was variable apart from the input and output layers. Each new chromosome created by the genetic algorithm was passed to a neural network generator to create the network which was then tested for its fitness on the robot. Tournament selection was used, although rather than using crossover which can be disruptive to the chromosome, they used mutation only as well as a change length operator which could add or delete neurons. As the chromosome was capable of increasing in size and thus increasing the search space, the chromosome size was included as part of the fitness function. A penalty was given to a large size, helping to limit the size of the artificial neural network. Several controllers were evolved including obstacle avoidance and garbage collection, where the robot had a gripper that would pick up an object and place it outside the arena.

Berlanga et al. [118] used evolutionary strategies to evolve an artificial neural network for a robotic controller. The experiment used a feed forward artificial neural network to control a simulation of the Khepera robot for navigation and collision avoidance. The inputs to the artificial neural network were the infrared sensors and wheel encoders, while the outputs from the artificial neural network were connected to the wheel drivers. A simulation of the Khepera robot was developed and used to test the process. The artificial neural network neurons were encoded as a twenty dimensional real-valued vector which were evolved using evolutionary strategies. Although it was found that the evolved artificial neural network adjusted its weights rapidly to the training environment, it did not perform well when tested in an environment that was different to the one it trained in.

Lund et al. [23] combined a simulated and physical implantation of an evolvable robotic controller using an artificial neural network to control the navigation of a Khepera robot. Using its own sensors, the Khepera robot was able to map out its environment which could be used by the simulation. The artificial neural network was a simple two layer network connecting the sensors to the motors.

Evolving artificial neural networks in hardware (FPAA, FPGA) for robotic controllers

Gallagher et al. [119] have evolved a continuous time recurrent artificial neural network on a custom built hardware platform. A recurrent artificial neural network has discrete sequences due to the feedback occurring within the network allowing each of the neurons to compute its output simultaneously. In comparison a continuous time recurrent artificial neural network is a recurrent artificial neural network, where the inputs and outputs are not steps but a continuous time variable and the neurons have a temporal response. In the research undertaken by Gallagher et al., the neuron was implemented in hardware using a row of analog adders for the weighted inputs and leaky integrator, while an operational amplifier was used for the sigmoidal output from the activation unit. The experiment used a microcontroller to interface between the computer performing the genetic algorithm and the artificial neural network. The microcontroller changed the setting of the digital potentiometers. The neural network was evolved to control a legged robot.

Berenson et al. [68] used FPAAs to create a two layer artificial neural network that was used to control both a biped robot and a damaged quadruped mobile robot. Both controllers were evolved and evaluated on a physical robot without simulation. The neurons were created using a summing network for the weighted inputs and an integrator to create the threshold trigger level. The weightings, integration constants, and polarity of outputs were evolved by evolutionary algorithms. Elitism was used for selection on the population size of 32 while both crossover and creep mutation were used for reproduction.

Rocke et al. [120] showed how three neuron models that could be evolved by a genetic algorithm were implemented in a FPAA. The neuron models implemented were: a) the McCulloch-Pitts, with a step function on its output; b) the multi-layer-perceptron, where the threshold function was replaced by either a sigmoid, tanh or Gaussian curve; and c) the spiking neuron, where the output from the neuron was a spike which gradually decayed to zero over time.

Manjunath and Gurumurthy [121] investigated fabricating an artificial neural network on an integrated circuit, using special purpose configurable analog blocks CAB with differential feedback. They proposed using a pair of transistors for a synapse, with a current mirror for the signed weights associated with a neural network, and a

logarithmic amplifier for the activation function of the neuron to produce a sigmoid response.

Roggan et al. [122] used a FPGA to create a spiking neural network to control a Khepera robot for obstacle avoidance. The artificial neural network was comprised of spiking neurons classified as cells, whose inputs and outputs were spikes from other cells. The weightings and connectivity (neural pathways) could be dynamically changed at run-time by the control function input of each cell. This cell function and connectivity was defined in a genetic chromosome which was evolved using a genetic algorithm. A NIOS processor was used to control the evolutionary process, program each cell function, and interface between the sensors, motor control and the spiking neural network. An array of eight x eight neurons was created and evolved for obstacle avoidance. The fitness was evaluated on an individual that could maintain maximum forward speed and distance from an object with minimal rotation of the robot.

Hannan Bin Azhar and Dimond [123] evolved an artificial neural network on a FPGA for the navigation of a Kephera robot using a RAM base neural network. The genetic algorithm evolved an artificial neural network chromosome. This chromosome was held in RAM, determined the size of the neurons, the number of neurons per class, the number of classes, how the sensors connected, and the speed control for the motors. Thus the artificial neural network architecture and its behaviour were controlled by the chromosome. The chromosome also held the robot ID and its fitness evaluation which was used by the evolutionary process.

Amaral et al. [124] created a neuron on a programmable analog multiplexer array as two blocks. The first block was a body circuit block, which implemented the synaptic weights and summation. The second block was the activation function block, which implemented the activation function. The programmable analog multiplexer used a 128 bit configuration bit string to configure its analog circuits, and it was this bit string which was evolved using the genetic algorithm. The selection process was steady state, the population size was 400, and the reproduction used one point crossover with mutation. Amaral et al. successfully evolved both the body and functional activation.

4.2 Evolution of Fuzzy Logic Robotic Controllers

4.2.1 Fuzzy Logic Controller Overview

Fuzzy logic is used in control systems that utilize imprecise input data to determine what output action to take. The concept of fuzzy logic for control systems was created by Lofti Zadeh in the mid 1960's. At that time control systems were developed using precise (crisp) data of either true or false. Zadeh noted that as humans control systems using imprecise (fuzzy) data, so too could computer systems.

Boolean logic has only two states, true and false, with no input or output able to occupy both states at the same time. In comparison, fuzzy logic has a range of conditions related to the input and output states. For example the crisp data description of the temperature in a control system would state the input temperature was either hot or cold, or give a precise temperature such as 42 degrees. Fuzzy data however, would be imprecise in its description of the temperature, giving a range of conditions such as very cold, cold, warm, hot, and very hot. This is shown in Figure 4-5.

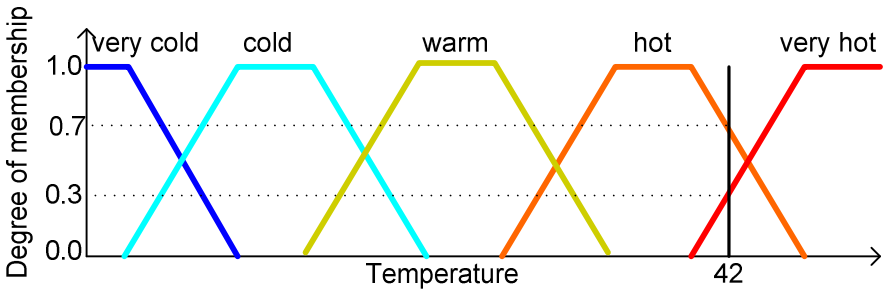


Figure 4-5. Membership function and degrees of membership for temperature inputs.

The input membership function is the shape illustrated in the above diagram, showing the magnitude of each input for the input range. The shapes in the above diagram are trapezoidal and shoulder, however many shapes can be used, such as bell or triangular. The degree of membership (truth value) is how much the input conforms to each pattern. In the diagram above, the degrees of membership for a temperature of 42 degrees are very cold 0.0, cold 0.0, warm 0.0, hot 0.7 and very hot 0.3. The output membership functions are described in a similar way with the heater element being nearly off, partly on, and almost fully on. The inputs from the sensors come in as crisp data, for example the temperature is 42 degrees. This crisp data is then converted to fuzzy data using the membership sets and the degrees of membership. This conversion is known as fuzzification of the inputs. In a similar manner the output of the fuzzy logic would be

converted from fuzzy data back to crisp data that can then be used by a control system. This is known as defuzzification.

Fuzzy operators are used to combine the fuzzy inputs into a value that the fuzzy rules can use. Three common operators are the AND, OR and NOT. These are graphically shown in Figure 4-6 where max(very cold, cold) acts like an OR of the two inputs, min(warm, hot) acts like an AND of the two inputs, and NOT(very hot) will invert the input. There are many other operators that are used to combine the fuzzy data.

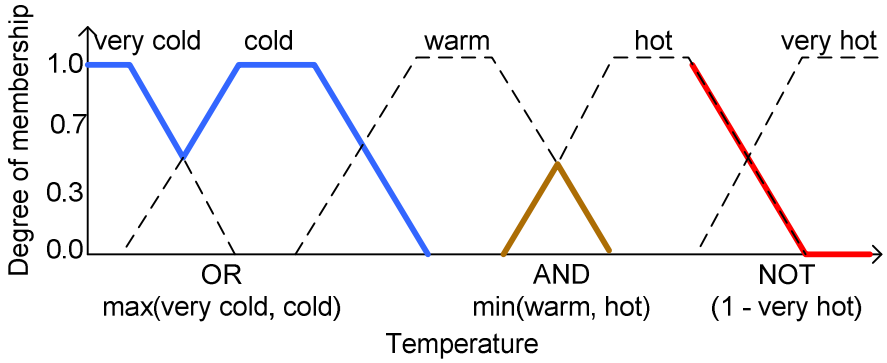


Figure 4-6. Graphical representation of fuzzy operators AND, OR and NOT.

Fuzzy logic is based on linguistic behaviours which are defined in the fuzzy rule base that relates the input sensor states to the desired output action states. Each rule in the rule base will activate (fire) with a strength proportional to the fuzzified input antecedent. These fuzzy rules are based on the if-then structure (else is not used as it will exclude ranges). The fuzzy rules describe what action to take for a range of input conditions. Rather than using precise measurements, the code could read 'if the temperature is cold and getting warmer then turn the heater on slightly'. The general form is **if** *variable* is *condition* (antecedent) **then** *action* (consequent). Note the antecedent and consequent are not absolute, that is if the antecedent is only partially true (less than one) then the consequent will have a corresponding degree of truth.

Thus the steps in fuzzy logic are: a) fuzzify the inputs: change the crisp data to fuzzy data using the membership functions to obtain the degree of truth between zero and one; b) inference and fuzzy rules: apply fuzzy operators if there are multiple inputs to give the antecedent a value between zero and one then use the rule base to produce the corresponding consequent; and d) defuzzify the outputs: pass the consequent through the output membership functions to produce a crisp output. These steps are shown in Figure 4-7.

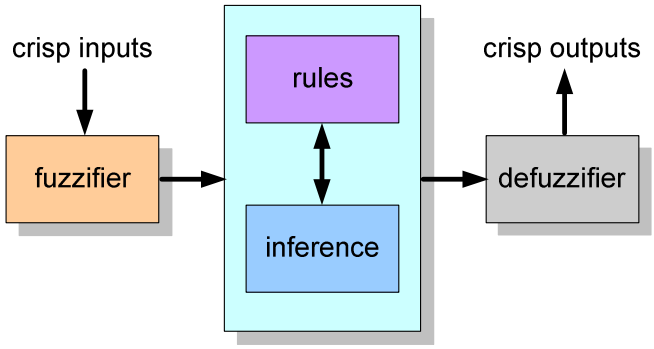


Figure 4-7. The three steps in a fuzzy logic controller.

4.2.2 Evolving Fuzzy Logic Controllers

The parameters of a fuzzy logic controller can be grouped into three areas. These are the fuzzification of the inputs, the fuzzy rules, and the defuzzification of the outputs. These areas are normally configured by experiment or design; however a genetic algorithm can be applied to these parameters to evolve the fuzzy logic controller. These parameters can be represented by a chromosome in several ways. A chromosome for the membership function is shown in Figure 4-8. The parameters in the chromosome show the shape descriptor: ls (left shoulder), tp (trapezoid), tr (triangle) and rs (right shoulder). The parameters immediately after the shape descriptor show the size of the shape giving the points at which the shape changes relative to zero on the x-axis. The fuzzy rules can also be encoded either as numerical values or symbols that are translated into linguistic rules.

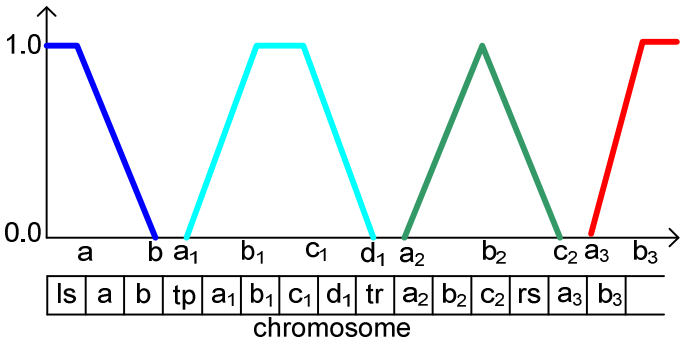


Figure 4-8. Chromosome representation of membership functions.

4.2.3 Examples of Evolving Fuzzy Logic for Robotic Controllers

The following section provides examples of evolved fuzzy logic robotic controllers that have been used in robotic applications.

Makaitis [66] used evolutionary strategies to evolve the fuzzy rule base in a fuzzy controller that was used to control the motion of a lift. The fuzzy rules were encoded into a chromosome by using integer numbers that represented linguistic terms such as a) position: including far above, above, same height, below, far below; b) velocity: including quickly upward, upward, slowly upward, stopped, slowly downward, downward, quickly downward; and c) acceleration: including high upward, upward, same speed, downward, high downward. These parameters were defined as real numbers. These linguistic terms were inserted into a Mamdani type rule such as 'if the position is far above, and the velocity is stopped, then set the acceleration to high downward'. The evolved controllers were tested in simulation, with the fitness determined by the time taken for the controller to bring the lift to a desired floor while following boundaries of maximum speed and velocity. It was found that the evolved controller was superior to custom designed controllers.

Sung Hoe et al. [125] used a genetic algorithm to evolve the fuzzy input and output membership functions as well as the rule base of a robotic fuzzy logic controller for a simulated wheeled robot. They used a variable length chromosome which allowed both the number of membership functions and rules to increase. The evolutionary process was performed in three stages: 1) the first step was to evolve the elements of the output membership function and the rule base that connects to them; 2) the removal of elements and rules that were not required while still retaining the same fitness; and 3) the input membership functions were evolved.

Jeong and Lee [126] successfully evolved the rules in a fuzzy logic controller that provided co-operative behaviour between multiple robots playing a predator-prey game where several robots worked together to capture a prey. This experiment used distributed artificial intelligence to develop co-operation strategies where the behaviour of each predator was governed by the fuzzy logic controller with all predators containing the same rule base. The rule base was a series of if then statements relating the current heading of the robot, the distance and angle to the prey, and the position of the other predator robots to the required robot direction. The robots were implemented in simulation with a fitness test ending after either the prey was captured or a set number of time steps were completed. The population size was 50, with each chromosome encoded with a 64 bit binary string, representing five linguistic variables, and two fuzzy sets for each variable; the selection process was roulette wheel.

Doitsidis and Tsourveloudis [127] investigated the role of the fitness function when applying a genetic algorithm to a fuzzy logic robotic controller with current fitness functions focused on the robot's completion of moving to a target with obstacle avoidance. Three types of fitness function were investigated: 1) aggregate (how well the robot completed the task without regard to how the task was achieved); 2) behavioural (how well the robot functions when performing a task); and 3) tailored (which had aspects of both aggregate and behavioural traits). The aggregate fitness was measured as to how close the robot moved to a target position. The behavioural fitness measured whether the robot moved directly to the target, and the tailored fitness measured both how close and how straight the robot moved to the target. The inputs to the fuzzy logic controller were the heading error, and the distance from the obstacles derived from two infrared sensors with the outputs driving the robot's left and right servo motors. The input membership functions were trapezoidal, whereas the output membership functions were triangular. These membership patterns were described in the chromosome. As the fuzzy rule base was fixed, it was not part of the chromosome. The evolution was performed on a real robot, with the evolutionary process stopped after 80 generations. Each generation was limited to 30 seconds of motion. From the researcher's experiments, it was determined that tailored fitness function produced the best result.

Gu and Hu [128, 129] evolved a reactive behaviour based fuzzy logic controller for a Sony legged robot that could play soccer, by evolving the input and output membership functions with a genetic algorithm. The required behaviour was to move towards the ball and face the goal. The inputs were the orientation of the robot relative to the ball, the distance of the robot from the ball, and the orientation of the goal. The output was discrete commands that could be recognised by the Sony controller. The fitness was a combination of the final position of the robot, and the least number of steps taken to move to that position. The population size was fifty, with one individual describing a complete fuzzy logic controller. The robot was tested in simulation and the evolution was completed after 300 generations. It was found that a fuzzy controller could be successfully evolved.

Li et al. [130] created a fuzzy logic controller that could park a simulated car into a garage. The novel concept was that they implemented the controller on a FPGA. Six input sensors for car position produced a kinematic model of the car. There were four stages to the parking process: 1) approach parking space; 2) pass parking space; 3) back

into parking space; and 4) final correction of park. Two controllers, one for steering angle and the other for speed control, were used in this experiment. They used the relative distances between the front of the car, the front right wheel, the rear right wheel, the front left wheel, and the rear left wheel as inputs to the fuzzy logic controller.

Wu et al. [131] evolved a fuzzy logic controller for a robot whose task was to navigate down a pipeline. The pipeline robot used fifteen ultrasonic sensors mounted on its side and front to determine its environment. Two wheel encoders were used to determine the robot position from a known starting point, and a speed sensor was used to determine the robot's velocity. These inputs were combined into three memberships: 1) distance from the robot to a wall; 2) the steering angle of the robot to a target; and 3) the robot's velocity. The outputs from the controller drove the robot's wheels. The robot's direction was determined by the shape of the pipe it was moving in (which was either straight, crossroad, t-junction or dead end). The rule base was comprised of linguistic variables that used six variables for the distance, three for the steering angle, and four for the velocity, giving 72 control rules which were encoded in the chromosome. In addition, the chromosome contained another 48 positions to describe the triangular membership functions. The fitness was comprised of three criteria: 1) the time taken to reach the goal; 2) the number of collisions of walls and obstacles; and 3) where the rules and fuzzy sets were kept to a minimum. It was found that the evolved fuzzy logic controller could control a simulated robot that could successfully find a good path.

Chronis et al. [132] used a genetic algorithm to evolve the rule base of a fuzzy logic controller whose task was to move towards a target while avoiding obstacles. The membership functions were kept fixed while the antecedents and consequents along with the number of rules in the rule base were evolved. The rules were of the general form: if obstacle distance is x, and obstacle direction is y, and target direction is z, then the robot direction is w. The chromosome that described the rule varied proportionally to the antecedent value of each variable with distance described in terms such as very-close, close, far, very-far, and direction described as front, front-right, right, back-right, back, back-left, left, front-left. The chromosome was divided into parts showing the total number of rules and the appropriate antecedent and consequent for each rule. The evolution was performed in simulation and the evolved chromosome was successfully tested on the real robot.

Islam et al. [133] created a fuzzy logic traffic controller that was implemented in hardware. The behaviours and the finite state machine were implemented in hardware using a hardware descriptive language. The inputs to the system were traffic volume

This chapter has reviewed both artificial neural networks and fuzzy logic controllers. The use of genetic algorithms to evolve these controllers has been examined.

Chapter 5

Chapter 5: A Review of Robotic Controllers for the Mobile Inverted Pendulum and Ball-Balancing Beam

Two robots were used to evaluate the robotic controllers that were evolved in this author's research. These robots included the mobile inverted pendulum and the ball-balancing beam. Both were chosen as they had a high degree of instability, making the control systems more complex. In the case of the ball-balancing beam, the beam was curved to make the system more unstable. This chapter reviews the current research associated with both the mobile inverted pendulum and the ball-balancing beam including their control systems, and control systems created by evolutionary computation.

5.1 Mobile Inverted Pendulum

Mobile inverted pendulums are of interest to university research due to their high level of instability and their application to robotics, such as walking gaits for bipedal robots. The reduction in the cost of gyroscopes and accelerometers has enabled universities and students to create mobile inverted pendulums in projects and for research. The most widely known implementation of the mobile inverted pendulum is the Segway used as a transporter for people and materials.

The Segway has been used in research including NASA's 'Robonaut'. Ambrose et al. [134] who worked for NASA, combined their robonaut upper torso with a Segway mobile base to give mobility to the robot and to enable human robot interactions such as following people, and tracking people with flashlights. Other planned uses were assisting astronauts in space, bomb disposal, and security.

Browning et al. [135] used the Segway robotic mobility platform that could interact with human players riding Segway transporters, to play a modified form of soccer. The robot used cameras for visual tracking to identify the ball, and a wireless interface to a central computer for position and game playing decisions. The referee communicated to

the human players via a whistle and verbal instructions, and to the mobile robots via wireless communications.

5.1.1 Non-evolved Robotic Controllers for a Mobile Inverted Pendulum

This section provides a review of the robotic controllers for the mobile inverted pendulum developed by other researchers.

Standard control state-space equations for a mobile inverted pendulum were developed by Grasser et al. [136] in their creation of JOE, the mobile inverted pendulum. JOE had three degrees of freedom, roll, yaw, and pitch, and was implemented as a non-autonomous robot. The control system that they constructed was implemented as two separate state-space controllers, the first acting on the yaw or turning motion, the second acting on the pitch or balance of the pendulum. The outputs of these systems were translated into signals for the right and left wheeled motors respectively.

Noh et al. [137] modelled a biped walking robot that used a balancing weight similar to an inverted pendulum using a linear non-homogeneous second order differential equation to find the zero moment point at the foot of the robot. Various gaits were produced in simulation to give the robot a walking motion.

Kim et al. [138] developed a two wheeled inverted pendulum robot that could be used as a home robot. They gave particular regard to the stability of the robot on inclined surfaces and turning motion. The researchers performed a detailed mathematical analysis of the kinematics of the robot on a flat surface and then advanced their models for inclined surfaces and finally for turning motion. They then used their models on a real two wheeled inverted pendulum with successful results.

Huang et al. [139] created a fuzzy controller for a two wheeled inverted pendulum using three fuzzy control units: the first for motionless balance, the second for travelling forward and backward, and the third for steering using yaw in the pendulum. All three fuzzy controllers were implemented by a NIOS processor inside a FPGA. The motionless balance fuzzy rules were based on a range of factors including the pendulum angle, angular velocity, the wheel angular velocity, the motor speed and the pendulum position. The travelling fuzzy rules were designed to move the pendulum to an unbalanced state so that it could move forward or backward, while the steering fuzzy

rules were designed to change the yaw of the pendulum so that it could steer left or right. Both simulation and real life experiments showed that the controller behaved well.

5.1.2 Evolved Robotic Controllers for the Mobile Inverted Pendulum

Anderson [140] described a method of using an artificial neural network to balance a inverted pendulum using reinforcement and temporal-difference learning. The only feedback mechanism for learning was a fail due to the pendulum falling over, thus the artificial neural network had to deal with delayed performance evaluation, as the pendulum could have a long sequence of actions before it fell over, making it difficult to determine which action led to the failure. He created two, two layered networks. The first was an action network with two outputs which drove the cart either left or right. The second was an evaluation network, which was used to determine which sequence of actions led to a failure. To overcome the delayed response in learning, the evaluation network used temporal-difference learning, which learns associations between signals separated in time, (in this case the states of the pendulum and the failure signals). Thus the evaluation network would indicate how soon a failure would occur. The output of the evaluation function was fed into the inputs to the action network along with the pendulum states. For example a failure would be likely to occur when the pendulum angle was near its boundary and the angular velocity was high.

Pasero and Perri [141] created a FPGA based neural controller used to balance an inverted pendulum on a cart, using an offline supervised trained multilayer perceptron. They used a hidden layer neural network, with five inputs: the pendulum angle, the pendulum angular velocity, the cart speed, the cart acceleration, and friction. The network had one output to drive the cart motor forward and backward. The artificial neural network used SRAM to store the weights, multiplication factors, number of inputs, number of neurons and activation function for each neuron. As a low resourced FPGA was used, only one neuron was constructed and time multiplexing was performed to provide for the number of neurons required.

Jung and Kim [142] used a neural network interfaced to a PID controller as shown in Figure 5-1, to balance a real mobile inverted pendulum. The neural network had six inputs, nine hidden layers and six outputs. The inputs to the neural network were the current and previous states of the pendulum angle θ and its horizontal movement x. The activation function within the network was the hyperbolic tangent function. The outputs

of the neural network were fed into two PID controllers, one to control angle, the other to control position, which were then combined to drive the two wheels of the motor. Both the neural network and PID controller were implemented on a TMS320F2812 digital signal processor.

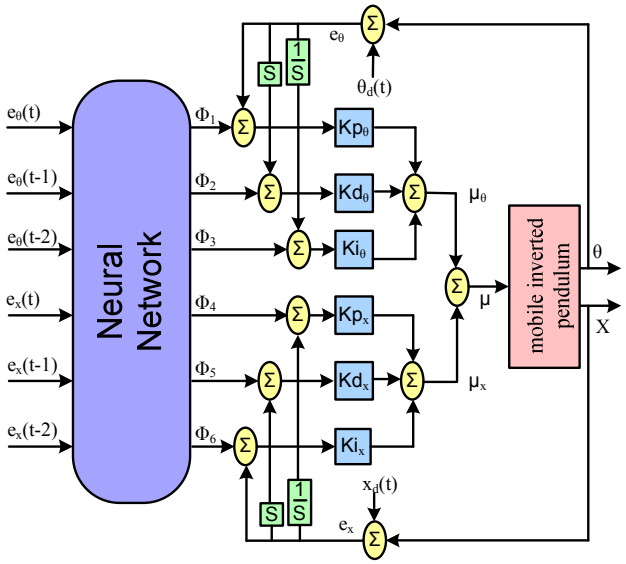


Figure 5-1. A neural network with PID for a mobile inverted pendulum.

Noh et al. [143, 144] created two robotic controllers using a similar technique to Jung and Kim, combining an artificial neural network with PID to create a controller that could balance the pendulum while it moved in a circular path. They used a radial basis function neural network with one hidden layer and a nonlinear radial basis activation function. The inputs to the neural network and connections to the PID controller were similar to the work performed by Jung and Kim.

Obika et al. [145] evolved a controller for an inverted pendulum with a double jointed arm as shown in Figure 5-2, which was capable of swinging its arm to an upright position and then keeping the arm upright. The chromosome was a variable length chromosome that was comprised of quantised motor speeds for set periods of time. The fitness evaluation was set by how quickly the pendulum would become upright, and how little movement there was in the arm once it was balanced. The genetic algorithm was based on the minimal generation gap model, which used a steady state algorithm and had a low selection pressure. The population size was ten, with the chromosomes starting with an initial length of 160 steps giving an eight second length of actions. The controller was successfully evolved, and compared well to a standard control system based on the zero dynamics method.

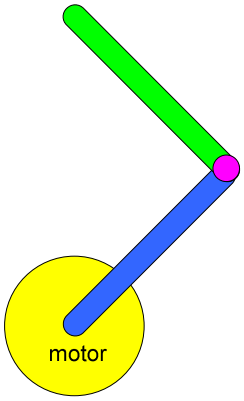


Figure 5-2. Double jointed inverted pendulum.

Hoffman [146] evolved a fuzzy logic controller for a pole balancing cart. The input and output membership functions along with the rule base were encoded into a chromosome that was evolved using a genetic algorithm. The chromosome was fixed in length, thus the number of membership functions and rules were fixed. The complete system is shown in Figure 5-3. The fitness was proportional to the amount of time that the beam remained balanced. The genetic algorithm used two point crossover with creeping mutation while the selection process was scalar.

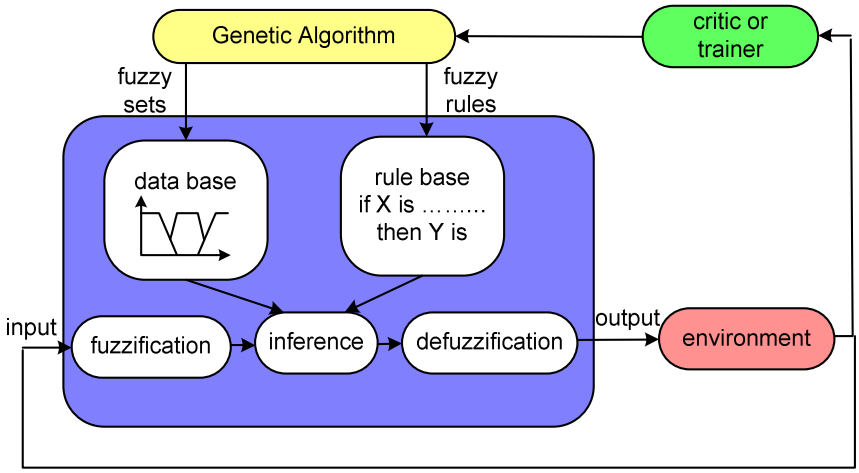


Figure 5-3. Block diagram of a fuzzy logic genetic algorithm.

Shieh et al. [147] used a genetic algorithm to evolve a Sugeno-type fuzzy logic controller for an inverted pendulum mounted on a cart. The Sugeno-type generates fewer rules than a standard fuzzy logic controller, making it more suitable for a genetic algorithm. Four fuzzy rules each having six coefficients were developed relating to the angle and angular rotation of the pole. The coefficients of the fuzzy rule base were encoded into the chromosome. The population size was twenty with crossover and a mutation rate of 0.5%. Two experiments were performed, one where the fitness

evaluation was determined by how quickly the pole would balance, the second for the reduction in overshoot as the pole was brought to balance. Both routines performed well.

Kwon and Lee [148] evolved a fuzzy logic controller that was used to balance an inverted pendulum mounted on a cart using evolutionary strategies. They used a Takagi-Sugeno type fuzzy logic controller with a Q-learning algorithm for its fuzzy rules. The chromosome was a string of real values used to configure the 25 rules in the rule base. Rank selection was used with two types of crossover, simple and arithmetical.

5.2 Ball-Balancing Beam

The ball balancing beam has been used as a standard laboratory experiment to demonstrate control systems for many years. It has also been employed as a benchmark for research into control systems due to its non-linear dynamics and behaviour. Control systems such as standard control, fuzzy logic, neural networks, and other systems have used the beam to test their responses.

5.2.1 Non-evolved Ball-Balancing Beam Controllers

Xuerui et al. [149] used a beam driven by two magnetic actuators to study how active magnetic actuators could be used to drive a tilt mechanism as shown in Figure 5-4. In particular they wanted to study the use of active magnetic actuators in the application of artificial blood pumps where the heart pump impeller was suspended in the pumping tube by magnetic suspension units. The difficulties in the control system were due to the non linear properties of the magnetic actuators. The objective was to keep the beam stationary around a nominal point. It was found that using an integral sliding mode controller with an integrator allowed the beam to reach a stable condition within 0.3 seconds of an external disturbance.

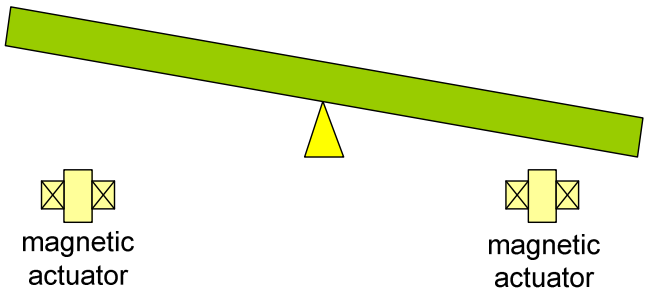


Figure 5-4. Beam controlled by magnetic actuators.

Ka and Nan [150] used a ball-balancing beam as a demonstration of optimal and disturbance accommodating control. To present a more difficult control system, the researchers used a beam with an arc which was mounted on a cart, as shown in Figure 5-5. The position of the ball was controlled by the horizontal motion of the cart which could be driven forward and backward by a DC motor; note the beam did not tilt. The aim of the control system was to keep the ball positioned at the centre of the arc while keeping the cart positioned at a midway point. They modelled the system using linear equations of motion, and used a linear quadratic regulator controller to stabilize the ball.

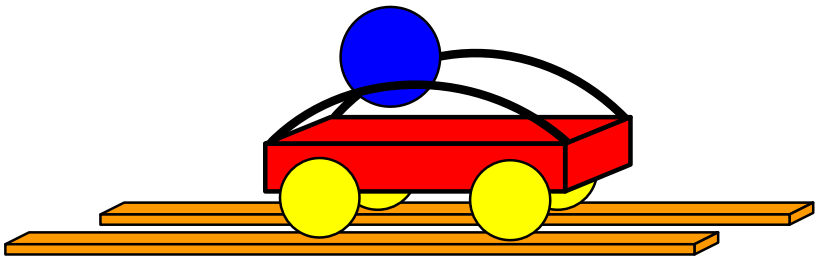


Figure 5-5. Ball-balancing beam on a cart.

Gordillo et al. [151] looked at the ball-balancing beam control system and how it would respond to transient disturbances. They used an asymptotically stabilizing controller to keep the ball stable on the bar during the transient conditions. They then created a controller that would cause the ball to oscillate between two fixed points.

Dadios et al. [152] incorporated camera vision and a fuzzy logic controller to balance the ball on a flat beam. The camera was used to accurately determine the ball velocity and distance of the ball from the centre of the beam. The fuzzy controller inputs were the ball velocity, the distance of the ball from the centre of the beam (taken from the camera), and the beam angle derived from a rotational sensor. These parameters were converted to a fuzzy input set and passed to the fuzzy controller inference and rule base. The output of the fuzzy controller was the motor speed and direction for the beam's DC motor. It was found that the beam could stabilize the ball in less than six seconds.

In a similar manner Iqbal et al. [153] created a fuzzy logic controller for the ball-balancing beam using a 68HCS12 micro-controller which is unique in the fact that it has inbuilt fuzzy logic instruction sets. A camera was used to determine the position of the ball using pixel images to determine the ball centre. This information was sent to a micro-controller and then used to calculate the error in the ball position (distance from the centre) and the rate of change in the error position. These three components (ball

position, ball error, and rate of change of ball error) were fed into the fuzzy inputs of the 68HCS12 where they were compared against membership functions to determine the degree of membership, and processed by the fuzzy rules. The resulting crisp data output was fed into a servo motor for beam control. The complete system is shown in Figure 5-6.

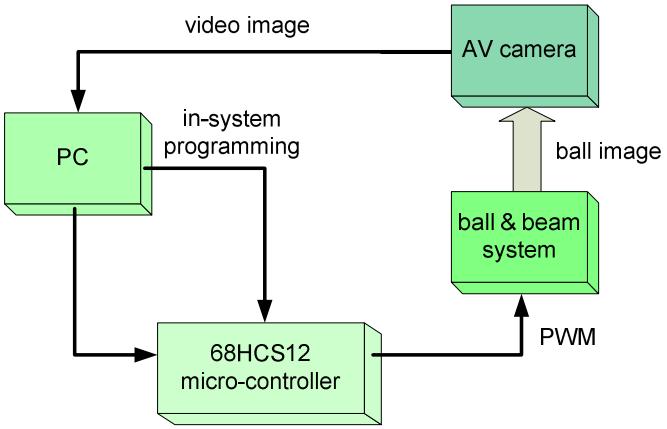


Figure 5-6. System interconnections for a ball-balancing beam using a 68HCS12 microcontroller with in-built fuzzy instructions.

Ng and Trivedi [154] combined a fuzzy logic controller in combination with a neural network for a ball-balancing beam. The system as shown in Figure 5-7 was comprised of three sections: 1) a fuzzy membership function, where the three inputs ball velocity, ball position and beam angle were fuzzified using triangular functions; 2) a rule neural network which mapped the fuzzy input vectors to fuzzy output vectors; and 3) an output refinement neural network which was used to drive the motor. Using the neural networks allowed a reduction in the number of fuzzy if-then rules that were required, thereby allowing more tolerance in the input parameters and the ability to cope with a noisier system. Both neural networks were multilayer feedforward with a back propagation algorithm. The system was able to balance the ball using a range of different balls and masses.

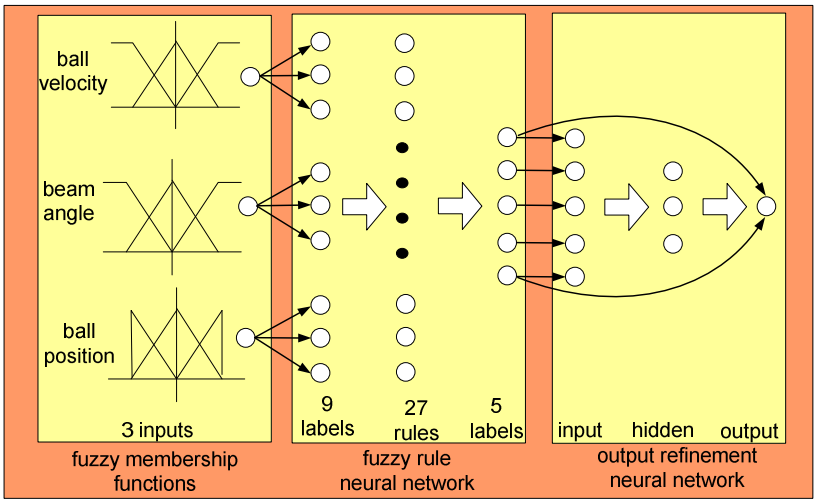


Figure 5-7. A fuzzy logic controller for the ball-balancing beam.

Eaton et al. [155] increased the complexity of the ball-balancing beam system by using a beam covered with a sticky substance, which gave an uneven response of the ball to the beam position. This made it difficult to model the system using classical control techniques. The researchers increased the difficulty to the controller by not using the ball velocity which is commonly used in other systems. They then developed a recurrent neural network using truncated back-propagation for the controller. The ball position, obtained from a single line 512 element CCD camera, and the beam position were fed into a Kalman filter to update the weights of the network.

Benbrahim et al. [156] used reinforcement learning on a connectionist actor-critic neural network to balance the ball. This type of neural network has two networks, the actor (action) network that controls the beam motor and the critic (value) neural network that provides reinforcement learning to the actor network. The four inputs to the neural network were ball position, ball velocity, beam position and beam angular velocity, while the output from the neural network provided the DC motor speed and direction.

5.2.2 Evolved Ball-Balancing Beam Controllers

Tettamanzi [157] evolved a ball-balancing beam fuzzy logic controller based on the SGS-Thomson fuzzy controller processor. The processor contained a weight associative rule processor, up to 16 inputs membership functions of any shape, up to 256 rules containing up to four antecedents and one consequent clause and 128 output membership functions. Five beam states were input to the fuzzy controller: ball position, ball velocity, ball acceleration, beam angle and beam angular velocity. The controller

had one output that was used to drive the beam motor. The chromosome for the neural network was the variables used to configure the SGS-Thomson fuzzy controller processor which were the input and output membership functions and the rule sets. As there were specific requirements for the set up of the processor, hand written chromosomes were used for the initial population and chromosome repair was required after the reproduction process. The evolution went through three stages, keeping the ball on the beam, keeping the ball in the centre of the beam, and finally moving quickly towards the beam centre and stability.

Yi and Xiuxia [158] used a genetic algorithm to successfully evolve a PID ball-balancing beam controller where the chromosome was a double float real number comprised of the three PID: constants, proportional gain, the integral constant and the differential constant. They used a modified form of genetic algorithm called the chaos genetic algorithm which mimics chaos theory where very small differences in an initial variable (termed the chaotic variable) causes large differences in long term behaviour. The aim of the chaos genetic algorithm is to help reduce premature convergence and reduce the number of iterations required to find a solution. The genetic algorithm used the island selection model with the population divided into separate groups.

This chapter has reviewed the use of mobile inverted pendulums and ball-balancing beams using non-evolved and evolved control systems.

Chapter 6

Chapter 6: Systems Developed for Experimentation

This chapter describes the common systems that were developed for the experimentation performed in this research. It is broken into sections:

- the mathematical models required for the simulations of the mobile inverted pendulum and the ball-balancing beam;
- the conversion of the mathematical model into simulation;
- the graphical user interfaces that were used for control, monitoring and data recording;
- the data command protocols used to connect the graphical user interface on the computer to the NIOS processor on the FPGA and then to the evolving virtual FPGA and hardware simulation.

The research presented in this thesis created two novel evolutionary capable robotic controllers and a hardware based simulation. The first evolutionary capable robotic controller was based on a lookup table; the second on a virtual FPGA. In order to evaluate these controllers and hardware simulation, two robotic platforms were chosen, both based on student projects performed at AUT University. The first was a mobile inverted pendulum; the second was a ball-balancing beam. These two robotic platforms are described in the next two sections.

6.1 Mobile Inverted Pendulum

6.1.1 Overview of the Mobile Inverted Pendulum

An undergraduate student project performed at AUT University was the design and construction of a non-autonomous mobile inverted pendulum which was capable of balancing upright and controlled motion using PID control software. The hardware was comprised of an Atmel 8-bit microcontroller which interfaced to a digital gyroscope, a three axis accelerometer, wheel encoders and a RF receiver. The microcontroller

calculated the beam roll tilt and yaw parameters by applying a Kalman filter to the data received from the gyroscope and accelerometers. These values, along with the pendulum planar positions determined from the wheel encoders, were used to balance the pendulum in its inverted state. The pendulum was also capable of motion by adjusting its tilt and yaw until it was off balance, allowing a horizontal movement to take place for compensation. The mobile inverted pendulum was successfully built and tested, as shown in Figure 6-1, and became the platform used for the evolvable controller.

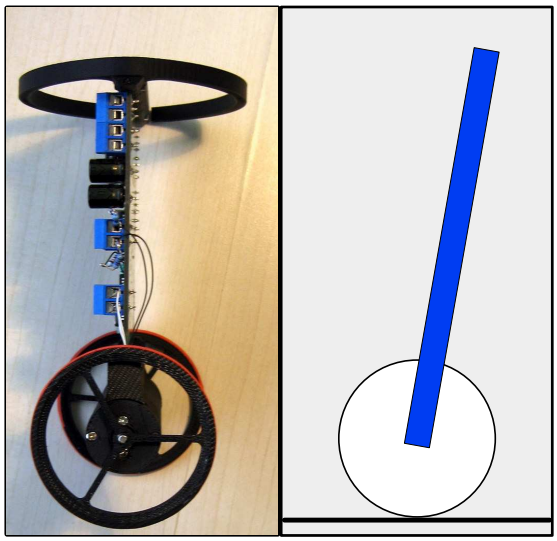


Figure 6-1. The physical and simulated mobile inverted pendulum used to evaluate the robotic controllers.

6.1.2 Mathematical Model of the Mobile Inverted Pendulum

The pendulum is a non-holonomic robot with three degrees of freedom (DOF), two planar motions and one tilt-angular motion, but with direct control of the pendulum in only the planar motions driven by the two wheels. Thus the control of the planar motion must work in such a way as to control the angular motion of the pendulum. As shown in Figure 6-2, the pendulum can rotate around the z axis (tilt); this is described by its angle θ_p and its angular velocity ω_p. The pendulum can move on its x axis described by its position x and its velocity υ.

The parameters used for the mathematical model are shown in Table 6-1.

g	gravitational acceleration (m/s^2)
θ_p	angle of the pendulum relative to the vertical axis (rad)
M_p	mass of the pendulum (kg)
M_w	mass of the wheel (kg)
Ø	angle of the wheel (rad)
I_p	inertia of the pendulum (kgm^2)
I_w	inertia of the wheel (kgm^2)
r	radius of the wheel (m)
x	horizontal displacement (m)
l	length from the axis to the centre of mass (m)
T	motor torque (Nm)
F	friction force (N)
H	horizontal force (N)
V	vertical force (n)
p	x coordinate of pendulum centre of mass
q	y coordinate of pendulum centre of mass

Table 6-1. Parameters used in the mathematical model of the mobile inverted pendulum.

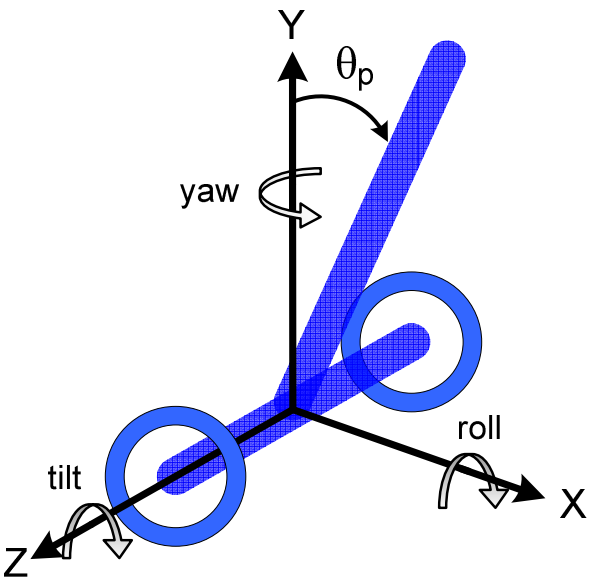


Figure 6-2 Diagrammatic sketch used for the mathematical model of the mobile inverted pendulum

To simplify the evolutionary process, the pendulum was constrained to only one planar axis x by driving the two wheels together so there would be no yaw. The pendulum was

on a flat surface so there would be no roll, therefore yaw and roll would not be implemented in the mathematical model. The important parameters were the pendulum's angle, angular velocity and the linear displacement along the x-axis.

Wheel

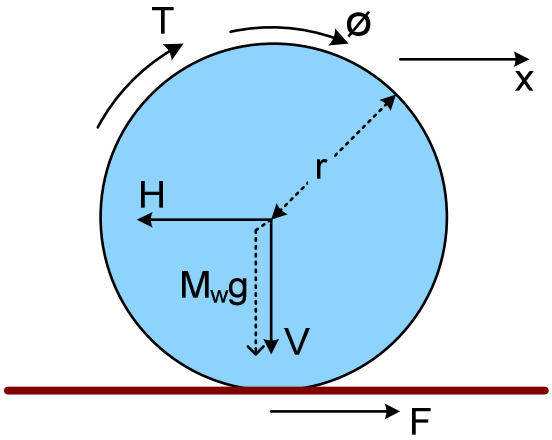


Figure 3. Pictorial representation of the torque produced on the wheel.

The motion of the wheel is described in Equation 6-1 (horizontal motion), and Equation 6-2 (rotational motion). The vertical motion is not used thus the ground reaction force in not needed.

$$M_w \ddot{x} = F - H \qquad \textbf{Equation 6-1}$$

$$I_w \ddot{\emptyset} = T - Fr \;\; = \;\; I_w \frac{\ddot{x}}{r} \;\; (as\ x = r\emptyset) \qquad \textbf{Equation 6-2}$$

Body

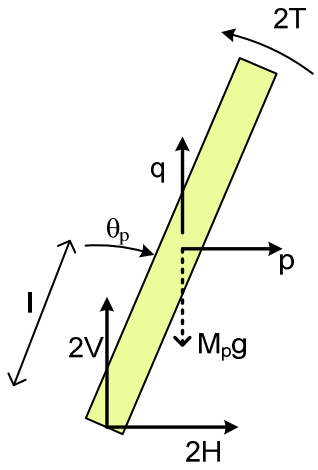


Figure 4. Pictorial representation of the forces on the pendulum.

The motion of the wheel is described in Equation 6-3 (horizontal motion), Equation 6-4 (vertical motion) and Equation 6-5 (rotational motion)

$$M_p\ddot{p} = 2H \qquad \textbf{Equation 6-3}$$

$$M_p\ddot{q} = 2V - M_p g \qquad \textbf{Equation 6-4}$$

$$I_p\ddot{\theta} = 2Vlsin\theta_p - 2Hlcos\theta_p - 2T \qquad \textbf{Equation 6-5}$$

Also

$$p = x + lsin\theta_p \qquad \textbf{Equation 6-6}$$

$$q = lcos\theta_p \qquad \textbf{Equation 6-7}$$

From the above equations the motion of the pendulums angle, angular velocity and linear displacement along the x axis can be described by Equation 6-8 and Equation 6-9.

$$\left(M_p + 2M_w + \frac{2I_w}{r^2}\right)\ddot{x} = M_p lcos\theta_p\ddot{\theta}_p = \frac{2T}{r} + M_p l\dot{\theta_p}^2 sin\theta_p \qquad \textbf{Equation 6-8}$$

$$\left(I_p + M_p l^2\right)\ddot{\theta}_p + M_p lcos\theta_p\ddot{x} = M_p lgsin\theta_p - 2T \qquad \textbf{Equation 6-9}$$

6.2 Ball-Balancing Beam

6.2.1 Overview of the Ball-Balancing Beam

The second robotic platform was a ball-balancing beam where the beam was curved, as shown in Figure 6-5. The ball-balancing beam was created by AUT students over several sequential final year projects. The beam was designed to demonstrate general control principals, such as a PID control. A stepper motor was used to alter the beam angle via a three to one right angled gear drive. The curved beam and the frame for the motor were hand built.

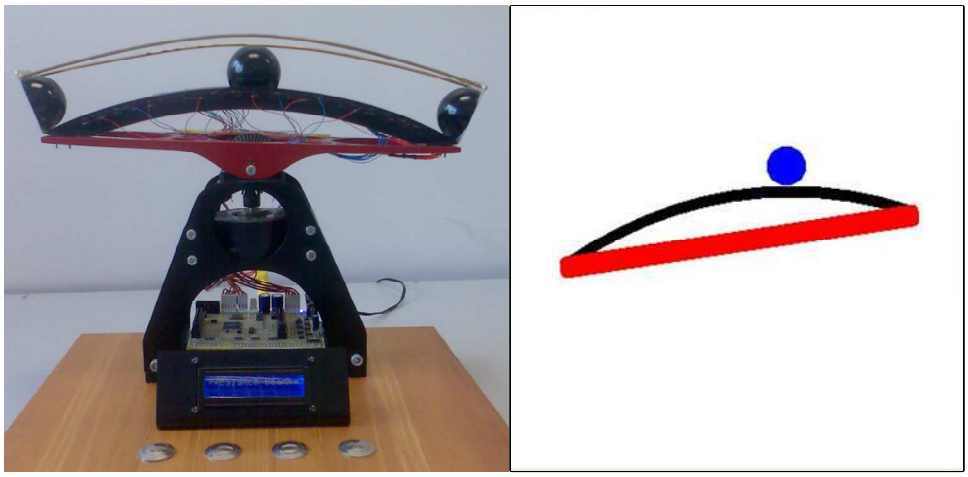


Figure 6-5. The physical ball-balancing beam system and the GUI display that the simulation controlled.

An Atmel mega128 microcontroller was used to control the motion of the beam, using a daughter board for the microcontroller and a mother board for the signal conditioning of the inputs and outputs of the beam. The position of the ball was determined by twenty one ball position sensors which used modulated infrared LED transmitters and photodiode receivers. It was possible to have two sensors activated at the same time, when the ball was between two sensors thus doubling the resolution of the ball position. Two limit switches mounted on the motor frame indicated when the beam was at either end of its travel. The original stepper motor had a maximum pulse rate of 8ms (125Hz) with each pulse producing a 0.22^0 shift in the beam, giving a maximum angular beam motion of 27.5^0/s. The maximum travel of the beam was 60^0 (30^0 to the left and 30^0 to the right), thus it took 2.2 seconds to move the beam from the maximum left position to the maximum right position.

Using standard proportional-integral-derivative control techniques, it was found that the motor was not powerful enough to move the beam quickly enough to balance the ball on the beam. Steps to overcome this problem would be to use a more powerful stepper motor, or to reduce the curvature of the beam itself.

6.2.2 Mathematical Model Ball-Balancing Beam

The parameters used in the mathematical model are shown in Table 6-2.

g	gravitational acceleration (m/s^2)
R	radius of curvature of the beam (rad)
m	mass of the ball (kg)
r	radius of the ball (rad)
I	rotational inertia of the ball (kgm^2)
θ	ball position (angle from the centre) (rad)
Ø	beam position (angle from horizontal) (rad)
F	frictional force (N)
P	reaction force (N)
W	weight force (N)
x	distance of the ball from the beam centre (m)
v	velocity of the ball along the tangent m/s
a	acceleration of the ball along the tangent m/s^2
ω	angular velocity of the rolling ball (rad/s)
T	torque on ball (Nm)

Table 6-2. Parameters used in the mathematical model of the ball-balancing beam.

In the model of the beam as shown in Figure 6-6, the beam position was measured as an angle φ (phi) from horizontal, and the ball position was measured as an angle θ (theta) from the centre of the beam.

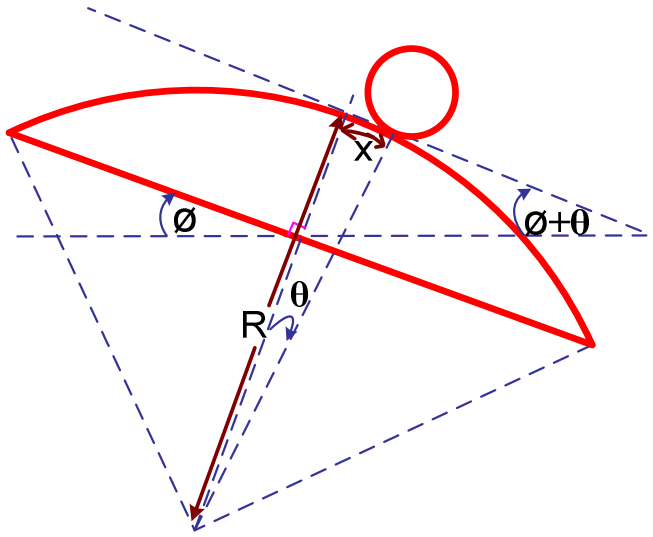


Figure 6-6. Diagrammatic representation of the angles θ and Φ in the ball-balancing beam.

In the free body diagram of the ball as shown in Figure 6-7, the ball is shown on the tangent to the beam. The three forces on the ball are its weight force W, the reaction force P of the beam on the ball and the friction force F. The direction of the friction force assumes the ball was travelling to the right, moving down the beam.

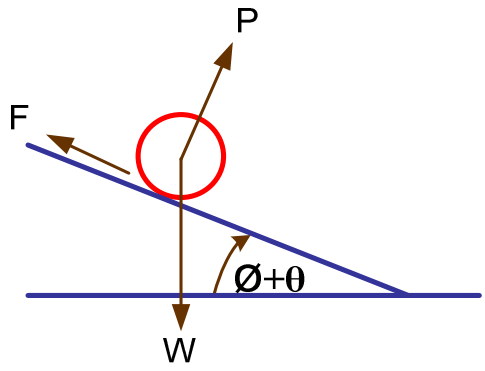


Figure 6-7. Diagrammatic representation of the three forces applied to the ball on a slope.

From Newton's second law of motion, the resolving forces parallel to the tangent gave the ball acceleration:

$$ma = -F + W\sin(\phi + \theta)$$ **Equation 6-10**

The forces perpendicular to the tangent are not used in this analysis. Assuming the ball was rolling, the rotational motion of the ball is given by:

$$I\dot{\omega} = Fr$$ **Equation 6-11**

The relationship between the ball's linear and rotational motion is:

$$a = r\dot{\omega}$$ **Equation 6-12**

Eliminating F and $\dot{\omega}$ and replacing W by mg:

$$ma = -\frac{Ia}{r^2} + mg\sin(\phi + \theta)$$ **Equation 6-13**

The relationship between the ball's distance from the centre of the beam and its angle from the centre is:

$$x = R\theta$$ **Equation 6-14**

so that

$$a = \ddot{x} = R\ddot{\theta}$$ **Equation 6-15**

Eliminating a between 4 and 5:

$$\left(m + \frac{I}{r^2}\right)R\ddot{\theta} = +mg\sin(\phi + \theta)$$ **Equation 6-16**

Hence

$$\ddot{\theta} = \frac{g}{R\left(1 + \frac{I}{mr^2}\right)}\sin(\phi + \theta)$$ **Equation 6-17**

This can be written as

$$\ddot{\theta} = A\sin(\phi + \theta)$$ **Equation 6-18**

where

$$A = \frac{g}{R\left(1 + \frac{I}{mr^2}\right)}$$ **Equation 6-19**

Note that A is a positive constant. When $\emptyset + \theta$ is greater than zero the ball accelerates to the right and when $\emptyset + \theta$ is less than zero, the ball accelerates to the left. This shows the inherent instability in the beam with the inverted curve.

If the angles θ and ϕ are small then

$$\ddot{\theta} = A(\phi + \theta) \qquad \textbf{Equation 6-20}$$

The stepper motor can directly control the rate of change of the beam angle ($\dot{\phi}$) which indirectly controlled the beam angle (ϕ).

For the actual beam *A* was found experimentally by rolling the ball down a stationary beam. *A* was found to be twelve.

6.2.3 Ball-balancing beam simulation mathematical model

To simplify the simulation calculations, Equation 6-20 was converted to the following units, corresponding to the parameters actually measured by the sensors on the beam.

Where

x - ball position from the position sensors (-19 to +19)

b – beam position from horizontal in units of the stepper motor pulses (-135 to +135)

v – ball speed = $\dot{x}$ (-1 to +1)

d – distance between ball sensors

δ - change of beam angle for a single pulse.

$$\theta = \frac{dx}{R} \qquad \textbf{Equation 6-21}$$

$$\emptyset = \delta b \qquad \textbf{Equation 6-22}$$

$$\frac{d\ddot{x}}{R} = A(\delta b + \frac{d}{R}x) \qquad \textbf{Equation 6-23}$$

$$\ddot{x} = Ax + \frac{A\delta R}{d}b \qquad \textbf{Equation 6-24}$$

$$\ddot{x} = a = 12x + 2.8b \qquad \textbf{Equation 6-25}$$

The simulation calculates

$$v_{new} = v + at \qquad \textbf{Equation 6-26}$$

$$x_{new} = x + vt + \frac{at^2}{2}$$ **Equation 6-27**

Substituting in the value for acceleration from Equation 6-25 and using a time period of 1ms

$$x_{new} = x + \frac{v}{10^3} + \frac{12x + 2.8b}{2\text{x}10^6}$$ **Equation 6-28**

Changing to integer with divisors to the power of 2

$$x_{new} = x + \frac{1049v}{2^{20}} + \frac{101x + 24b}{2^{24}}$$ **Equation 6-29**

Thus we can find the new ball position.

The ball velocity can be found from its current velocity plus its acceleration. Replacing acceleration with empirical data and using a time step of 1ms

$$v_{new} = v + \frac{12x + 2.8b}{10^3}$$ **Equation 6-30**

Changing so divisor is a multiple of 2

$$v_{new} = v + \frac{786x + 184b}{2^{16}}$$ **Equation 6-31**

Note the simulation's values chosen for the divide where carefully chosen to represent a number equating to a power of 2, this meant that the hardware description language synthesis could use left shifting or other minimization techniques, rather than a divide function. The RTL viewer in Quartus allows the user to see a schematic of the internal structure of the design net-list. An investigation of the hardware simulation that was generated by Quartus showed that no dividers where used in the circuit, and only five signed multipliers where used. The rest of the circuit was comprised of multiplexers and comparators.

Chapter 7

Chapter 7: Evolving Lookup Tables for Robotic Controllers

This chapter presents a novel approach of using a genetic algorithm to evolve a lookup table which was used as a controller for robotic applications. Experimentation was performed on two robotic platforms, the mobile inverted pendulum and the ball-balancing beam.

In chapter five, a literature review of the mobile inverted pendulum and the ball-balancing beam was carried out. This review described current research on the actual modelling of these systems and the control systems used to balance them. These control systems included PID, fuzzy logic and artificial neural networks. In the past, most of the research that used genetic algorithms to evolve the control systems for robotic applications was performed on evolving either an artificial neural network or a fuzzy logic controller. With regard to artificial neural networks, the genetic algorithm was used to evolve the weightings and neural pathways. In the case of a fuzzy logic controller, the genetic algorithm had been applied to the input and output membership functions and the fuzzy rules base.

As an alternative to using PID, artificial neural networks or fuzzy logic controllers for robotic control, the author of this research has taken the unusual approach of basing the controller on a multidimensional lookup table. The axes of the lookup table were connected to the robot's input sensors providing the current state of the robot, such as its position or speed. The parameter at each position within the lookup table gave the desired action that the robot should take, and this parameter was sent to the robot's actuators (for example a motor). In this way the lookup table could be used to provide an output which controlled the actions of the robot dependent on its input states.

If used in a conventional way, the use of a lookup table for a robotic controller would require the initialization of the lookup table with the appropriate parameters before use. These parameters could be derived from standard mathematical models and control algorithms, such as PID control for the robot. However in this application the lookup

table was initially loaded with random values and a genetic algorithm was used to evolve the appropriate lookup table parameters for the system to function.

Two robotic controllers, the mobile inverted pendulum and the ball-balancing beam, were evolved using a software genetic algorithm applied to a lookup table. A block diagram of the complete system is shown in Figure 7-1. The system contained a graphical user interface for a dynamic visual representation of the robot, control of the evolutionary process, and data logging to record the evolutionary steps. The outputs of the simulation presenting the current states of the robot were connected to the axis of the multidimensional lookup table. The parameters at the specified position within the lookup table were sent to the inputs of the simulation to control the actions of the simulated robot. The software genetic algorithm used the lookup table itself as a chromosome, and evolved a population of these lookup tables until a solution was found.

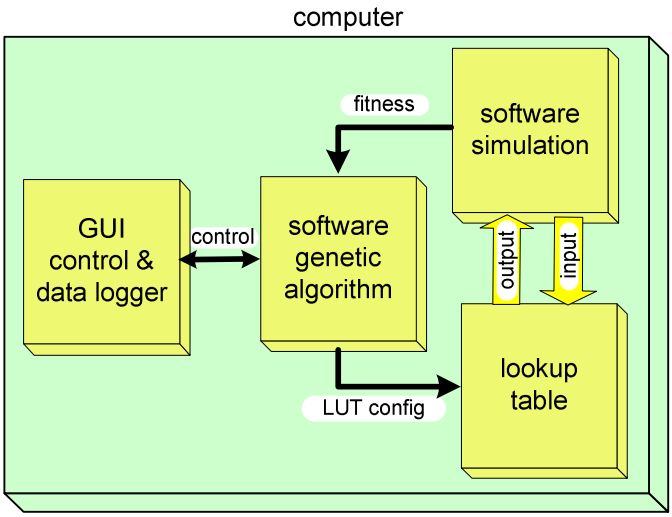


Figure 7-1. Block diagram of the systems and interconnections for the software genetic algorithm used to evolve a lookup table.

7.1 Evolving Lookup Tables for the Mobile Inverted Pendulum

The main difficulties in the field of evolutionary robotics are that:

- initial chromosomes can be destructive to the robot and its environment;
- initial chromosome populations have very little selective pressure as they all perform poorly making the beginning phase of the evolutionary process slow (this is known as the bootstrap problem [159]);
- robotic tasks are complex, creating a large search space and subsequently a large amount of time is required to evolve a controller.

Robotic simulation is used to overcome the first two issues of destruction and selective pressure. The problem of a large search space can be diminished by either using coding methods to reduce the chromosome size and thus the search space, or by using subsumption architecture where individual behaviours of the robot are evolved independently of each other before being combined together. In this application subsumption architecture was employed with balancing the first behaviour to evolve. Future behaviours such as navigation or autonomy could then be independently evolved.

This experiment used a genetic algorithm to evolve a controller for a mobile inverted pendulum. The experiment used subsumption behaviours, where layers of behaviour were evolved separately and then combined to create more complex behaviours. The first evolved behaviour was to keep the pendulum balanced while moving only on the x axis. This was achieved by keeping the drive to each wheel the same value, thus removing the yaw, and it was operated on a flat surface so the roll had been removed. The mathematical model of the mobile inverted pendulum moving on the x axis has been described in chapter six. The pendulum states that were used, were the pendulum angle θ_p, angular velocity ω_p, and horizontal position x. The following sections detail the graphical user interface, chromosome and genetic algorithm used to evolve the robotic controller.

7.1.1 Graphical User Interface

The graphical user interface shown in Figure 7-2 presents a diagrammatic representation of the pendulum and numerical displays of the pendulum's current state. The evolutionary processes such as current generation, individual number, average fitness and maximum fitness were also displayed. These parameters were automatically saved

to a file for later analysis. When enabled, the motion of the pendulum could be displayed in real time, although this slowed down the evolutionary process and was normally turned off. In a similar manner the pendulum and evolutionary parameters could also be displayed in real time. Both the best individual's fitness and average population fitness were automatically recorded at the end of every generation. In addition, the best individual chromosome and the motion of the pendulum could also be recorded. This motion showed the pendulum's angle and angular velocity, giving the positions in the lookup table that the individual stepped through as its fitness was evaluated. This data allowed the lookup table (chromosome) to be monitored as the chromosome was evolving, to see: a) what parts of the lookup table the individual passed through as it was evaluated; b) where it spent most of its time; and c) what caused the individual to fail the evaluation.

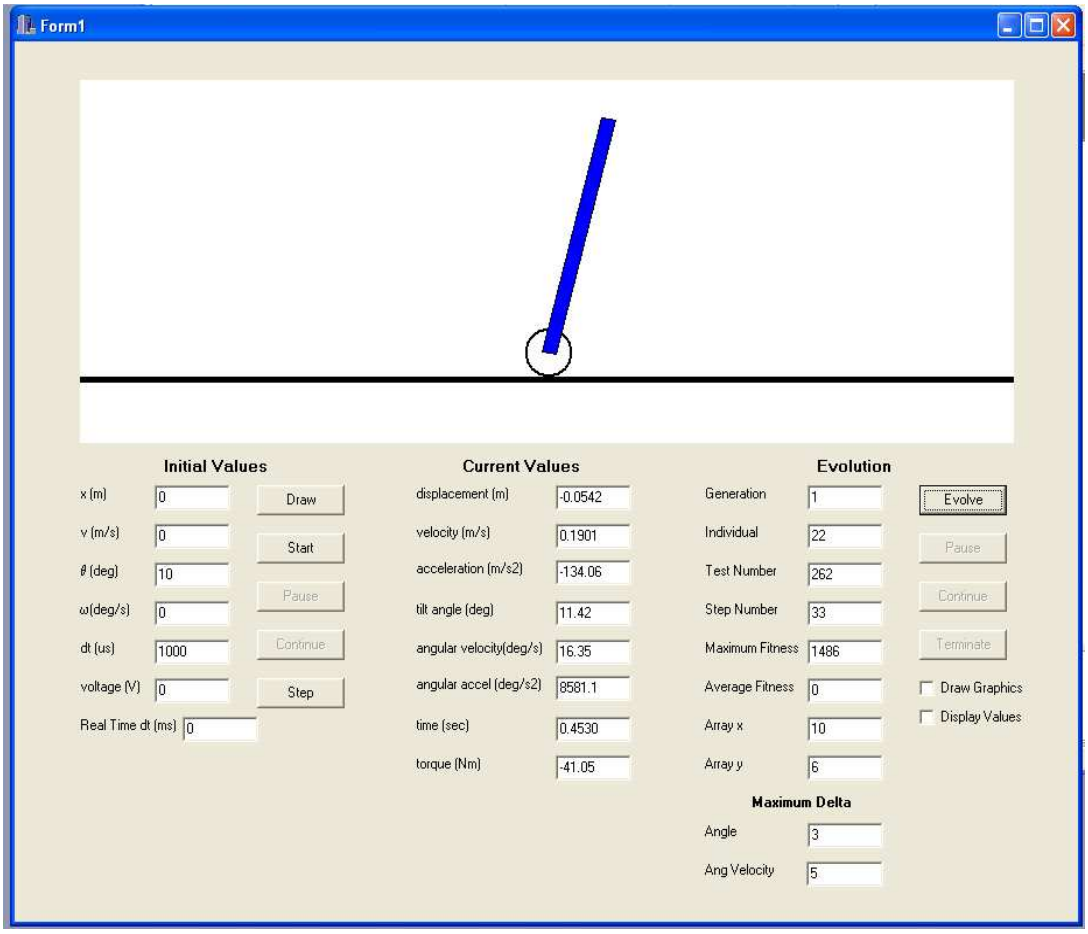


Figure 7-2. Graphical user interface used for the mobile inverted pendulum software genetic algorithm.

7.1.2 Genetic Algorithm

Chromosome

A chromosome is a possible solution to a problem. In regard to the pendulum, the chromosome was a two dimensional lookup table as shown in Figure 7-3, which related the angle and angular velocity of the pendulum to the required motor direction and torque. The columns represent the pendulum's current angle ranging ±18 degrees from vertical with a step size of 3 degrees. The rows represent the pendulum's current angular velocity ranging ±30 degrees per second with a step size of 5 degrees per second.

pendulum angular velocity (degress/second) \ pendulum angle (degrees)	-18	-15	-12	-9	-6	-3	0	3	6	9	12	15	18
30	250	25	75	200	50	225	100	75	250	25	175	150	225
25	50	125	175	0	200	150	225	100	0	75	250	200	100
20	0	200	125	0	150	50	100	250	25	250	125	0	75
15	250	25	225	150	50	25	100	50	200	100	250	75	200
10	0	50	200	250	75	75	150	75	50	200	200	0	225
5	25	200	100	250	25	175	100	175	0	75	100	150	200
0	100	0	50	200	150	50	175	250	150	25	100	75	50
-5	100	175	150	0	200	25	100	25	250	25	125	50	75
-10	250	25	175	250	175	250	0	175	150	0	175	75	250
-15	0	150	75	150	250	50	250	200	100	250	75	200	175
-20	175	0	250	200	175	50	100	25	250	250	100	100	200
-25	50	25	50	250	100	250	25	25	150	175	0	75	25
-30	175	200	200	0	250	125	50	150	225	150	75	200	250

Figure 7-3. Pendulum chromosome in the form of a two dimensional lookup table.

The simulated pendulum's output states of angle and angular velocity were linked to the two dimensional arrays column and row selections. The parameters inside the lookup table, which showed the required motor direction and speeds for a given angle and angular velocity, were connected to the simulated drive motor of the pendulum. The motor driver for the actual pendulum was an H-bridge driver controlled by an eight bit number. This number was a linear representation of the motor direction and torque, with 0 representing the maximum reverse torque, 125 representing the motor stopped and 250 representing the maximum forward torque. These values were mimicked in the chromosome. The step size was 25, allowing a maximum of 11 possible motor torque settings, 5 forward, 5 reverse and 1 stopped.

The search space for this chromosome can be calculated by Equation 7-1. This equation relates the number of positions in the lookup table (169 positions), with the number of possible speeds (eleven speeds), giving a total search space of $11^{169} = 9.9 \times 10^{175}$.

$$search\_space = number\_of\_speeds^{number\_of\_table\_positions} \qquad \textbf{Equation 7-1}$$

Reproduction

The chromosome reproduction used a two point crossover scheme where two points within the parent's chromosome were randomly chosen and the gene code of the two parents between these two points swapped to create two offspring as shown in Figure 7-4.

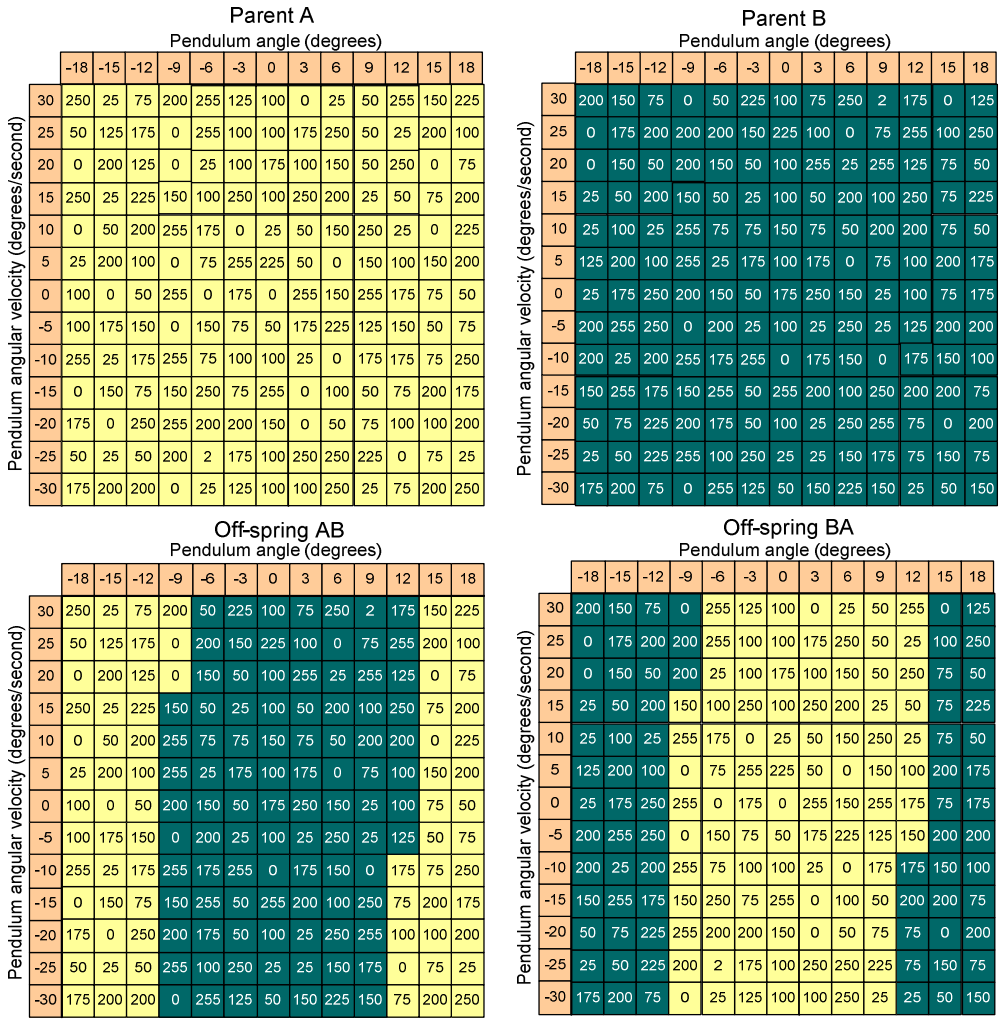


Figure 7-4. An example of two point crossover on the pendulum chromosome.

To generate two point crossover, four points were selected. These were the column and row at the start of the crossover, and the column and row at the end of the crossover. To find the first starting points, two random numbers between the values of one and

thirteen were generated. To find the two end points, two random numbers between the starting points and thirteen (the number of rows and columns) were generated. Creeping mutation was then applied to the newly generated offspring to maintain population diversity. Creeping mutation is a mutation technique where the gene is replaced with a value within a limited range of the original non-mutated value. On each generation after crossover was performed, five of the hundred individuals were chosen for mutation. A mutation of ten randomly positioned genes within the individual chromosome was performed.

Selection

The selection process used was tournament selection. This process divides the population into subgroups of individuals, and after fitness evaluation, only the fittest individual within that subgroup is retained. The selection pressure in tournament selection is dependent on the subgroup size. If the subgroup size is large then only a few individuals in the total population will be retained after each generation, thus the selection pressure will be high. However with a high selection pressure over time, there will be a corresponding drop in chromosome diversity, increasing the possibility of the evolutionary process becoming trapped in local maxima. As the size of the subgroup decreases, more of the total population of individuals will be retained. Thus the selection pressure will decrease, while the population diversity will be maintained. The selection process in this experiment had a subgroup size of two, giving low selection intensity while maintaining a diverse chromosome pool.

The genetic algorithm stepped through the population of individuals, selecting two individuals that were adjacent to each other and removing the individual with the lower fitness. When the selection process was completed the order of population was shuffled, allowing reproduction to occur from different parents after each generation.

7.1.3 Simulation and Coding Structure

The simulation used floating point numbers for the pendulum angle and angular velocity. These numbers were required to be converted to the x-y coordinates of the lookup table (zero to twelve). A flow chart illustrating the interaction between the simulation and the lookup table is shown in Figure 7-5. The simulation ran until an angle or angular velocity boundary was reached. The simulation then gave the current floating point values for the pendulum's angle and angular velocity to a subroutine,

which converted these parameters to the lookup table axis, to provide a new motor speed. A new angle and angular velocity boundary was generated and passed to the simulation.

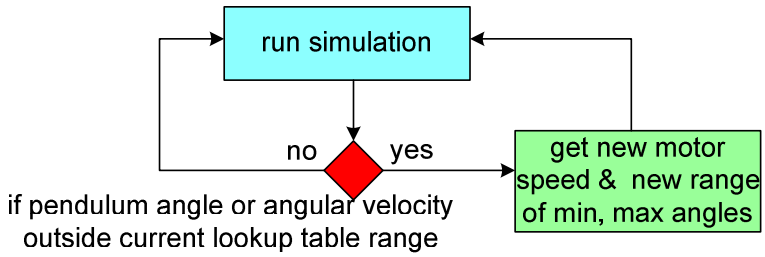


Figure 7-5. Flow chart of the simulation's interaction with lookup table.

The basic coding structure showing the iteration of reproduction, fitness evaluation and selection that is performed by the genetic algorithm is provided below.

```
main()
{
        initialise_population();
        record_parameters();  // store the maximum and average fitness, and generation
        do
        {
                procreate(); // perform crossover and mutation
                terminated = find_fitness(); // run simulation over a range of start conditions
                                             // can also record the pendulum motion
                best_fitness = selectiont(); // perform tournament selection
                if(generation%5 == 0)
                        record_parameters();  // store the max & av fitness, and generation
                        if (store_chromosome)
                                record_chromosome() ; //store the chromosome population
                        if(store_motion)
                                record_motion(); // store the motion of the pendulum
                generation++;
        } while (generation <= 500 && !terminated); //
        record_parameters();  // store the maximum and average fitness, and generation
}
```

7.1.4 Fitness Evaluation

The main component of the fitness evaluation was the length of time that the pendulum remained upright within a set angle. However after initial experiments, other fitness evaluation criteria were included, allowing the evolutionary process to produce an

increased performance of the pendulum with a more robust robotic controller. The final fitness was determined by the length of time that the pendulum remained within a vertical angle ranging from ±18 degrees and within a horizontal position ranging from ±0.5 meters of its start position.

Two experiments were performed, each with a different range of pendulum starting angles. The first had a pendulum starting angle ranging from ±18 degrees, with 12 starting angles each spaced 3 degrees apart. The beam was not tested at the vertical position as starting an ideal simulation of a pendulum at zero degrees and zero angular velocity with zero motor speed would give this starting position a perfect score. In real life this could not occur as there would always be a slight pendulum angle or angular velocity. The second experiment had a pendulum starting angle ranging from ±12 degrees with eight start angles each spaced 3 degrees apart. Once again no test was run with a pendulum starting angle of zero degrees.

The fitness evaluation of each individual was stopped after 300 seconds if the pendulum had not moved outside either the maximum vertical or horizontal range within this time. A time of 300 seconds (five minutes) was considered to be a thorough assessment, giving a high probability that the pendulum would remain balanced indefinitely. Each individual was tested twelve times from twelve different start angles; these times were summed and then divided by twelve to give the average time that the pendulum had remained balanced. This was performed in a similar manner for the experiments with eight starting angles ranging from ±12 degrees.

The process of determining the fitness criteria was modified during experimentation to improve the behaviour of the evolved pendulum. This fitness criteria were modified in the following manner. The original genetic algorithm had the pendulum starting from only one angle. An investigation on the evolved chromosome found that only a limited part of the lookup table would evolve. This occurred as the pendulum would learn to balance from its initial offset starting angle. However the pendulum did not learn how to balance from other initial start conditions. To prevent this from happening, the same individual was evaluated over 12 starting angles, ranging from ±18 degrees with a step size of 3 degrees. The starting angular velocity was 0.

After modification of the original fitness evaluation criteria, experimentation began again. A further study of the pendulum's horizontal motion found that the pendulum

would remain upright, however it would develop a constant forward or backward momentum, i.e. the pendulum would maintain a fixed balance point that required a constant horizontal motion moving the pendulum away from its starting position. It was determined that this was a poor characteristic and it was penalized by terminating the experiment when the pendulum moved beyond ± 0.5 meters from the initial horizontal starting position. Thus the fitness for each individual was now determined by both its ability to balance the pendulum and to remain stationary.

Further experimentation found that some fitness evaluations would last an indefinite period of time. This was a good result for one individual at one starting point, however the fitness evaluation would never end and the evolution process would stop. To overcome this, all fitness evaluations were terminated after a 300 second interval, thus the maximum fitness that the controller could achieve would be five minutes.

A final investigation of the lookup table found that the parameters at the extreme positions of the table did not evolve to the value expected from the mathematical model; with a motor speed at a maximum forward or reverse torque. An investigation of the pendulum's motion found that at the ranges of ±18 degrees the motor, even at maximum torque, did not have enough power to bring the pendulum upright. Thus the extremities of the lookup table would not evolve as there were no values that would offer a significant difference in fitness.

7.1.5 Results

Idealised lookup table

An ideal chromosome was derived from a standard control system algorithm with a linear progression of motor torque values dependent on the angle and angular velocity. When the pendulum angle and angular velocity was zero, then the motor would be stopped, with a motor parameter of 125. As the pendulum angle moved towards -18 degrees, the motor parameter moved towards 200 (nearly full left). As the pendulum angle moved towards +18 degrees, the motor parameter moved towards 50 (nearly full right). In both cases the motor was being driven in a direction that would return the angle of the pendulum to zero. A corresponding pattern occurred with the angular velocity. It was expected that the evolved chromosome would look like the ideal lookup table as shown in Table 7-1.

Angle (degrees) \ Angular Velocity (degrees/second)	-30	-25	-20	-15	-10	-5	0	5	10	15	20	25	30
-18	225	200	200	200	200	200	200	175	175	175	175	175	150
-15	200	200	200	200	175	175	175	175	175	175	150	150	150
-12	200	200	175	175	175	175	175	150	150	150	150	150	150
-9	175	175	175	175	175	150	150	150	150	150	150	125	125
-6	175	175	175	150	150	150	150	150	125	125	125	125	125
-3	175	150	150	150	150	150	125	125	125	125	125	125	100
0	150	150	150	150	125	125	125	125	125	100	100	100	100
3	150	125	125	125	125	125	125	100	100	100	100	100	75
6	125	125	125	125	125	100	100	100	100	100	75	75	75
9	125	125	100	100	100	100	100	100	75	75	75	75	75
12	100	100	100	100	100	100	75	75	75	75	75	50	50
15	100	100	100	75	75	75	75	75	75	50	50	50	50
18	100	75	75	75	75	75	50	50	50	50	50	50	25

Table 7-1. An example of an ideal pendulum chromosome.

This ideal chromosome was run on the simulation, but when assessed it performed poorly, failing the test after less than one second. A review of the recorded pendulum's motion showed the reason for failure was that even though the pendulum moved to an upright position and could achieve balance, it did not become vertical quickly enough to avoid a horizontal drift within the required ±0.5 meter distance from starting. From this it could be seen that a successful chromosome would require motor torque settings that quickly moved the pendulum to a vertical position, and maintained that vertical position while keeping the pendulum within the horizontal boundary. Further investigation found that the motor torque was not strong enough to pull the pendulum upright within the ±0.5 meter boundary from start angles of ±18 and ±15.

Two groups of experiments were performed: one with a pendulum starting angle ranging from ±18 degrees, the second with a starting angle ranging from ±12 degrees. The population size was 100 with the starting population randomly generated. During the evolutionary process three events were recorded and analysed. These were:

- the fitness and chromosome of the best individual within the population;
- the average fitness of the population;
- the pendulum motion.

Results starting angle from ± 18 degrees

A typical result is shown in Figure 7-6 with an initial large gap between the best and average fitness, and then a convergence between these values with each generation. The best fitness increased in large jumps as there was a step change in the best individual within the large population. However the average fitness of the population gradually improved until the complete population had approximately the same fitness. It was thought at this point that the population had converged and diversity had been lost, with only mutation producing new variations in offspring. However an investigation of the population's chromosomes found that diversity still existed with different patterns of pendulum motion still being performed.

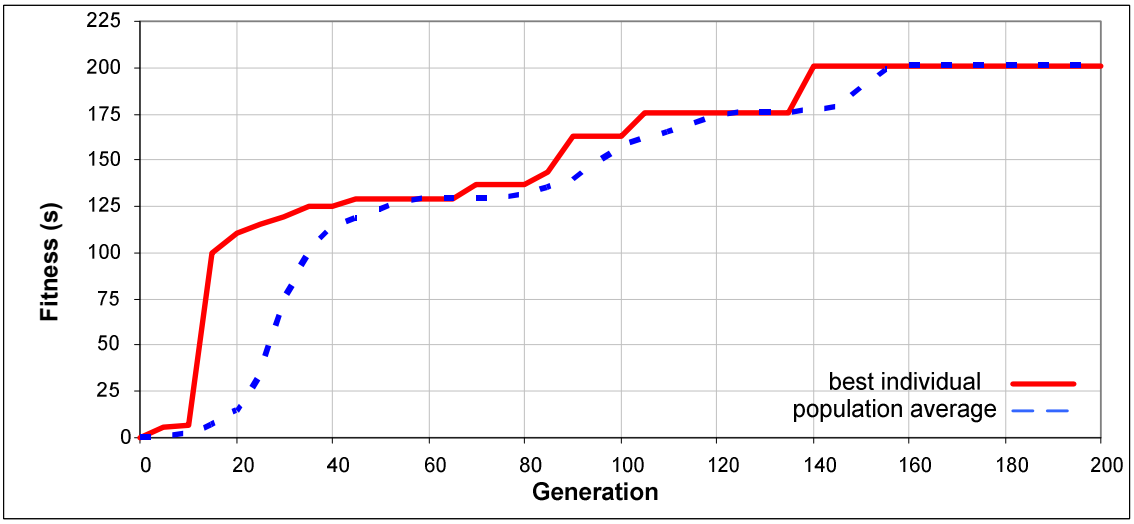


Figure 7-6. Fitness relative to generation for pendulum starting angle ±18^0 showing the best individual and population average fitness.

The best individual fitness and average population fitness of several successful runs are shown in Figure 7-7. It can be seen that the evolutionary process was similar over a number of experiments with the beam able to balance for 100 seconds within 40 generations, making steady improvement after that with the best individual eventually capable of balancing for 200 seconds. The upper limit for fitness was 300 seconds, although this was not reached because the pendulum could not start at an angle less than ±12 degrees without moving outside its ±0.5 horizontal position before it could become stable. This resulted in four of the test results behaving poorly, giving a maximum overall possible fitness of 208 seconds.

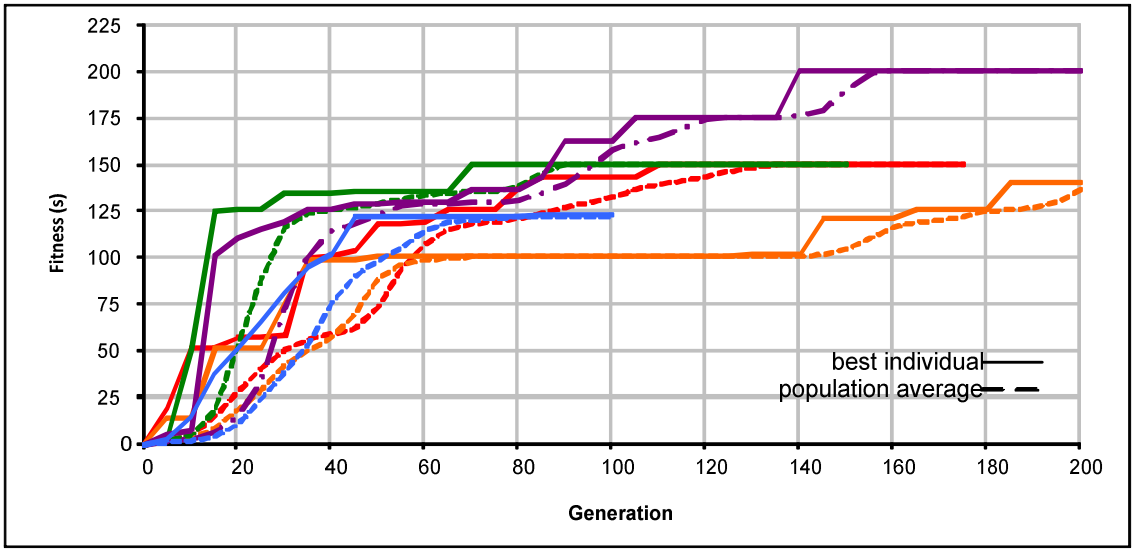


Figure 7-7. Fitness relative to generation for pendulum starting angle ±18^0 showing the best individual and average fitness for multiple runs.

Results starting angle from ± 12 degrees

The experiments were repeated with the starting pendulum angle ranging from ±12 degrees. The fitness step response of the best individual and gradual improvement of fitness for the population average was similar to the ±18 degree experiments as shown in Figure 7-8. The difference between the two was the maximum fitness that could be reached, changing from a maximum of 200 seconds to 262 seconds due to the reduced extreme starting angles of the second range of experiments.

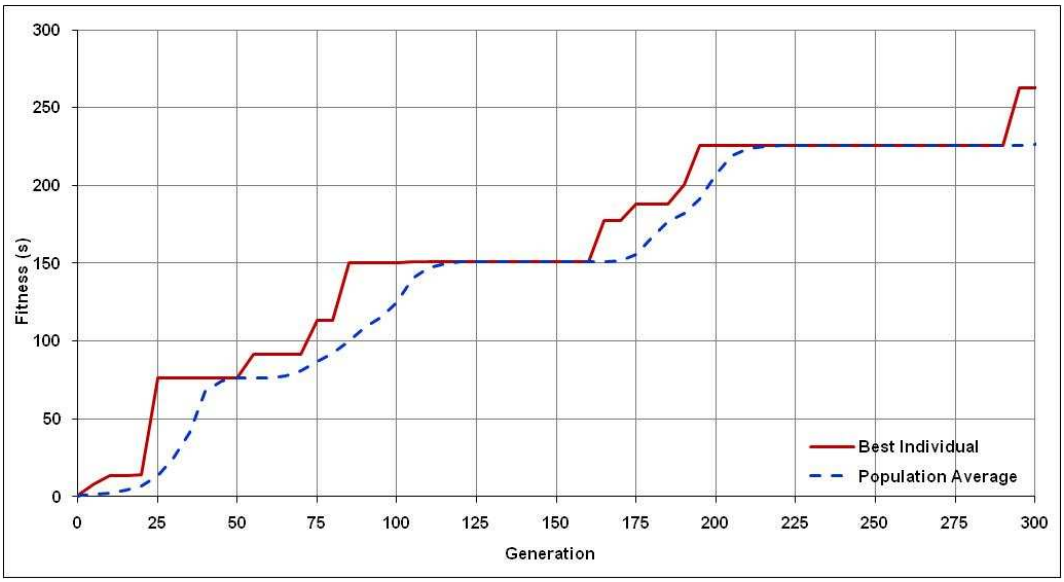


Figure 7-8. Fitness relative to generation for pendulum starting angle ±12^O showing the best individual and the population average.

Figure 7-9 shows the best individual fitness over several runs. It can be seen that the population has distinct stages to its evolution. The initial stage has a rapid increase in levels up to a balance period of 50 seconds and then step changes after that point. Then a plateau occurs at 150, 180, 220 and 262 seconds. These plateau are explained by the starting angles that are nearest to 0 degrees can be easily evolved, while the starting angles further away from the upright position require a large initial response to avoid the ±0.5 meter penalty. These are then evolved separately which are indicated by the large steps at these points.

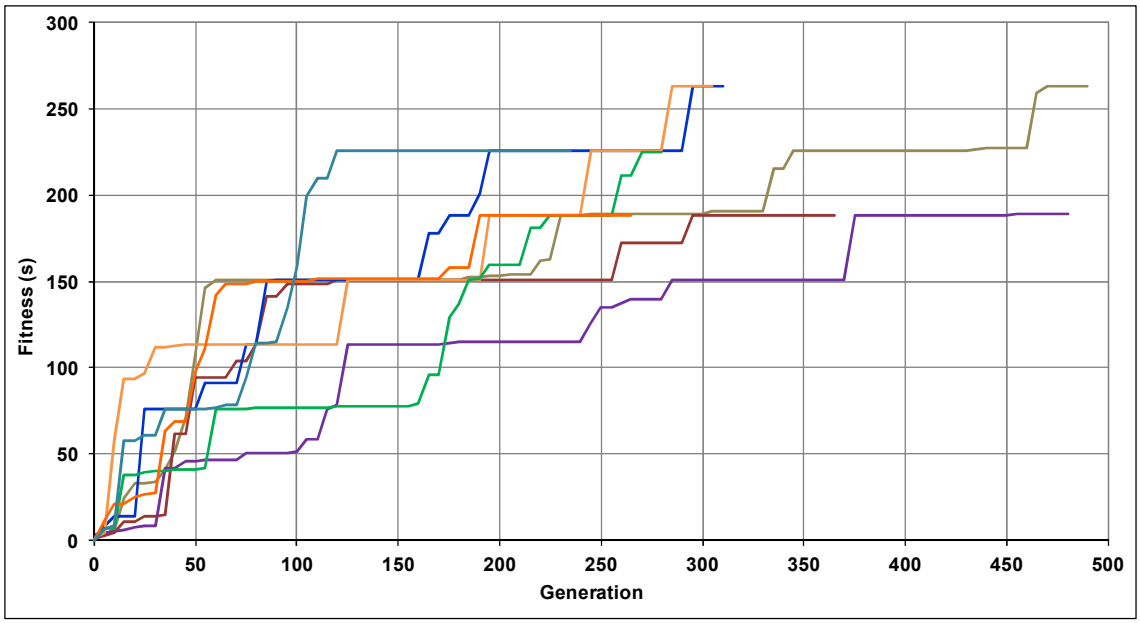


Figure 7-9. Fitness relative to generation for pendulum starting angle ±12^0 showing the best individual with multiple runs.

Figure 7-10 shows the best individual fitness and associated average fitness of the population over several runs. It can be seen that the best individual and average population fitness converge over 50 generations, then a resultant better fitness is found and there is a divergence in the best and average fitness. These eventually come together until the next leap in best fitness. An examination of the chromosomes indicated that the diversity was maintained, with a range of possible solutions being presented with different motions of the pendulum for each chromosome.

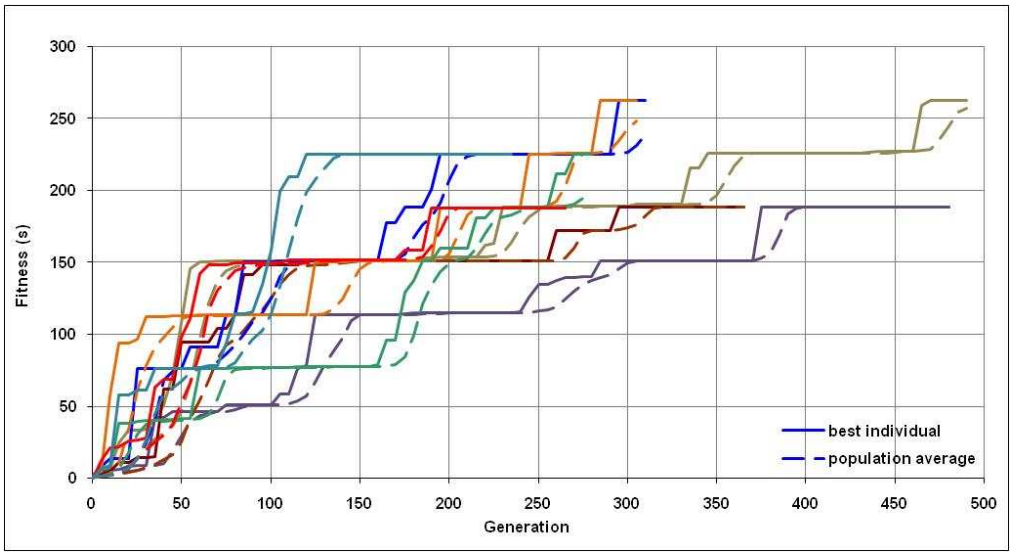


Figure 7-10. Fitness relative to generation for pendulum starting angle ±12^0 showing the best individual and population average with multiply runs.

Pendulum motion

The graphical user interface could record the motion of the pendulum during the simulation. This allowed the pendulum's access of the lookup table to be investigated showing the pendulum angle, pendulum angular velocity, and the corresponding motor parameters of direction and speed. The pendulum motion was analysed using a successful chromosome so the characteristics of the pendulum motion in relation to its chromosome could be observed.

The characteristics of a successful chromosome were that the pendulum would quickly be brought to an upright position, which prevented the pendulum from moving out of its ±0.5 meter horizontal limit. Two different characteristics of the pendulum motion were seen. Firstly, the pendulum would jitter around a static horizontal position so that horizontal drift was eliminated. Secondly, the pendulum would have a slow horizontal drift in one direction and then kick back to the start to begin the slow horizontal drift once again. It was noted that the pendulum did not use the entire lookup table; instead it would move through a set path which would be endlessly repeated. It was also observed that successful chromosomes differed from each other as there were many different possible means of successfully balancing the pendulum.

Interestingly the pendulum did not fail a test by exceeding its angular limit. Instead the termination of a test run was due to either the pendulum running to the time limit, or the

pendulum accumulating a small horizontal drift that over a long period of time would take the pendulum outside the ±0.5 meter horizontal limit.

A successful chromosome is shown in Table 7-2. An investigation of the parameters within the lookup table showed that at an angular velocity of zero, the motor was driven harder to the left as the angle moved towards -18 degrees, and harder to the right as the angle moved towards +18 degrees. This was the initial kick that the pendulum got at the beginning of its test.

The evolved chromosome had a non-linear progression between cells, thus the path that the pendulum took through the chromosome was highly convoluted. Some of the cells would seem to have incorrect values according to their angle and angular velocity; however adjacent cells compensated for the incorrect settings. It was this erratic sequence of motor torques that created the jitter and corresponding horizontal stability.

		Angular Velocity (degrees/second)												
		-30	-25	-20	-15	-10	-5	0	5	10	15	20	25	30
Angle (degrees)	-18	50	100	225	0	50	175	25	100	50	150	200	50	125
	-15	25	250	25	125	0	0	50	125	75	175	250	225	100
	-12	175	150	25	25	25	225	75	75	0	175	25	100	75
	-9	250	25	25	25	150	0	100	75	0	200	50	50	100
	-6	25	125	150	75	150	175	75	25	125	0	100	50	0
	-3	100	175	0	150	25	50	0	25	25	0	100	25	225
	0	225	100	100	150	25	250	75	150	200	50	100	200	100
	3	225	75	75	125	100	250	225	100	225	100	250	0	100
	6	200	225	200	175	225	225	0	200	175	100	0	150	25
	9	250	150	200	100	175	0	225	125	200	25	175	150	100
	12	225	225	175	200	225	125	225	225	150	200	25	225	150
	15	250	250	250	225	125	250	200	225	125	125	225	250	250
	18	50	250	50	50	75	150	250	250	100	0	175	200	200

Table 7-2. An example of a pendulum's evolved chromosome showing the relationship between angle and angular velocity with the motor speed output.

The best chromosomes from several evolutionary runs were compared and it was found that the chromosomes differed even though they had a similar fitness. This was due to the random initial chromosomes and the many possible successful solutions that could be evolved.

7.1.6 Conclusions

This first experiment has described a novel method in which a lookup table based robotic controller for a simulation of a mobile inverted pendulum controller could be evolved to a point where an acceptable level of balance was achieved. It was found that the genetic algorithm produced a controller capable of balancing the pendulum for 200 seconds within 200 generations (starting angle ranging from $\pm 18^0$) and for 262 seconds (starting angle ranging from $\pm 12^0$). The fitness evaluation was an important parameter of the evolutionary process as it determined the final behaviour of the pendulum. In the case of the mobile inverted pendulum, without a fitness penalty the pendulum would remain balanced but have an unwanted continuous horizontal motion. It was found that the population of individuals, though having the same fitness level had not lost diversity; instead multiple paths to obtaining a balanced pendulum were found.

A conference paper and a book chapter were published on using a lookup table to evolve a controller for the pendulum (see chapter 1). The simulation and graphical user interface can be found in the CD accompanying this thesis.

7.2 Evolving Lookup Tables for the Ball-Balancing Beam

The second experiment investigated the use of a genetic algorithm to evolve a ball-balancing beam controller by evolving a population of three dimensional lookup tables used to control the beam motor. The beam position, ball position and ball speed states of the beam were used to determine which parameter within the lookup table would be used to give the required motor speed and direction that would move the beam in a motion that would balance the ball. The genetic algorithm was similar to that used by the mobile inverted pendulum; however the search space was far larger due to the larger size of the ball-balancing beam lookup table. A graphical user interface was developed to record the beam and ball states, the chromosomes within the population, and the genetic algorithm parameters such as best fitness within the population and the average fitness of the entire population.

Historically a straight beam has been used for the ball-balancing beam, as it simplifies the control system algorithms that are required to balance the ball. However in this research the beam was curved, as this provided a more complex simulation model and algorithm, and also meant that the ball would never reach a static stable state with the motor stopped. The simulation was modeled around a ball-balancing beam that was developed at AUT University for a student project, as shown in Figure 7-11. The physical beam was curved; it had twenty-one infrared detectors to determine the position of the ball; and a stepper motor to control the angle of the beam. The angular velocity of the beam was controlled by the number of pulses fed into the stepper motor per second. The maximum angular velocity was determined by the maximum pulse rate that the stepper motor could respond to (125 pulses per second). The angular movement of 0.22 degrees per pulse gave a maximum angular velocity of the beam as 27.5 degrees per second.

The mathematical analysis for the ball-balancing beam has been described in chapter six. This analysis was converted into a simulation model that employed fixed integers that was used in the following chapters.

Figure 7-11. Picture of the ball-balancing beam developed in a student project.

7.2.1 Graphical User Interface

The graphical user interface is shown in Figure 7-12. This interface displayed the current ball position, ball speed, beam position, and the current time that the ball had remained balanced. A dynamic visual representation of the ball and beam in motion could also be turned on or off, allowing the user to see how the ball and beam were responding at various stages of the evolutionary process. The visual representation was normally turned off as when it was on, the evolutionary process was slowed to real time. Evolutionary control buttons were used to start, pause, and terminate the evolutionary process. The evolutionary parameters of generation number, current individual under test, average fitness of the population, and maximum fitness that has been reached was provided. A text display showed the number of speed settings, the maximum beam speed, the maximum fitness, average population fitness, generation number and time taken for the evolutionary process. These values were stored for later analysis.

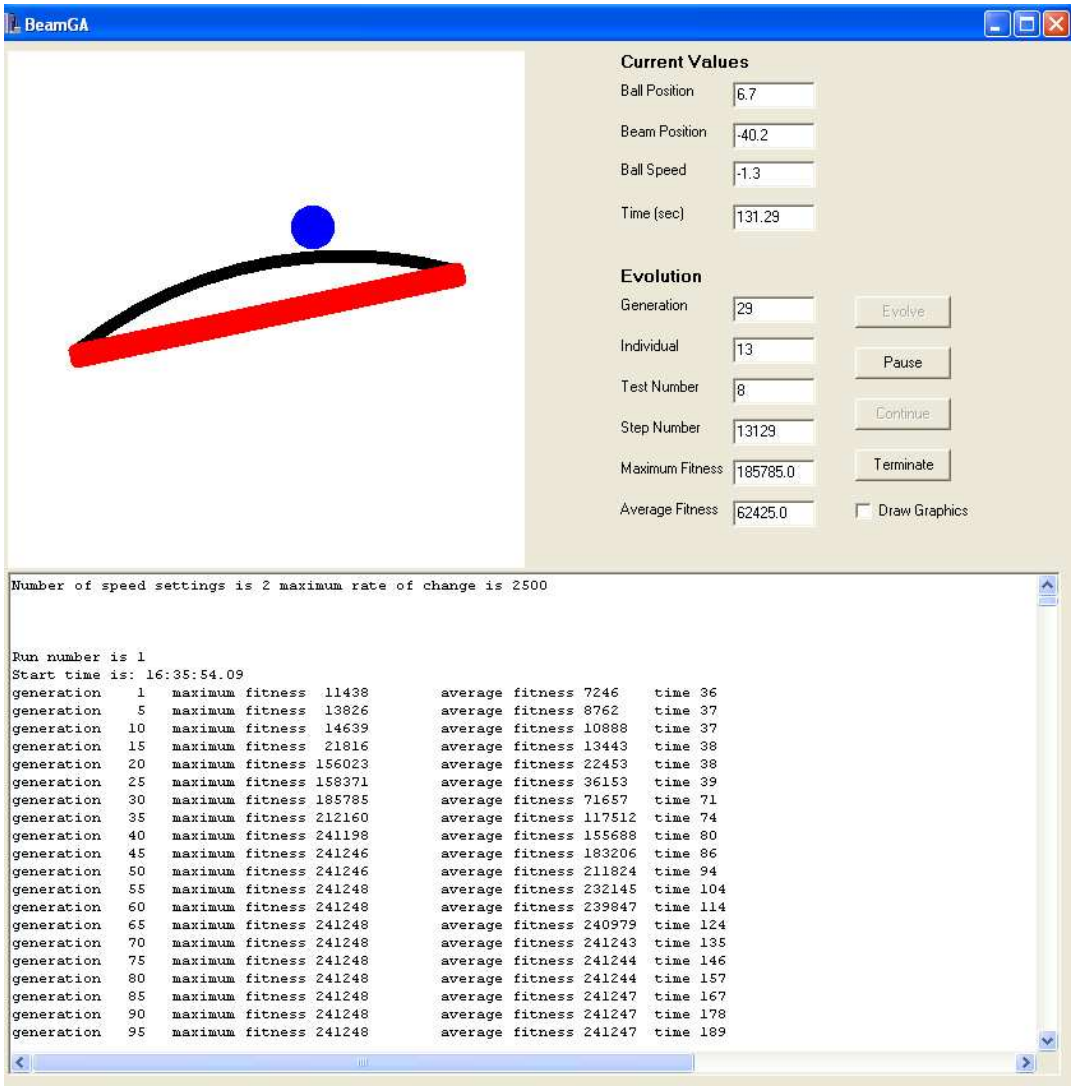


Figure 7-12. Graphical user interface for the ball-balancing beam controlled by an evolved lookup table.

7.2.2 Genetic Algorithm

Chromosome

The heart of the controller was a three dimensional lookup table as shown in Figure 7-13. The lookup table contained the desired motor speed and direction required to drive the motor in such a way as to balance the ball. The three dimensions of the lookup table were linked to the ball and beam states. These were, ball position (nineteen inputs), beam position (ten inputs), and ball speed (three inputs). Several lookup tables were evaluated with a range of motor speeds varying from two to eleven. The elements of the array were defined as char variables initialized with a randomly generated number quantised into 11 discrete steps ranging from 0 to 250. This enabled each location in the array to describe a motor speed with five left speeds, five right speeds and one stopped.

The speed range was reduced when evaluating different speeds by adjusting the threshold so that the motor had limited speeds. For example with a two speed range, values below 125 would drive the motor hard left, while values above 125 would drive the motor hard right.

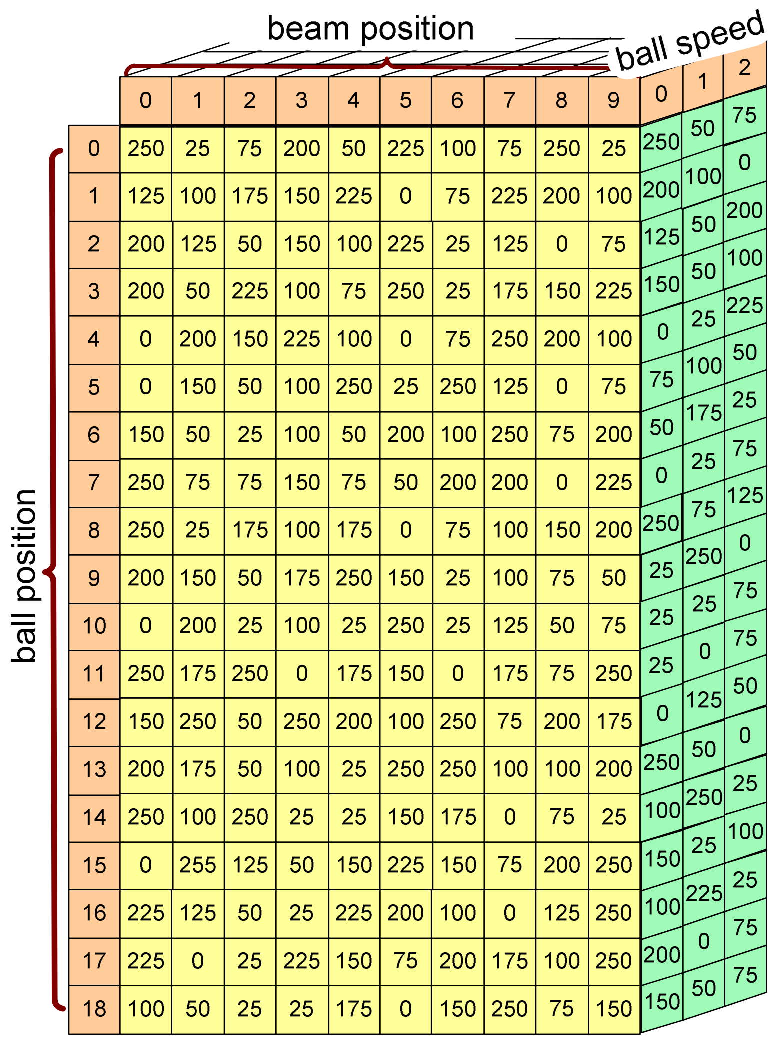


Figure 7-13. Balancing beam chromosome in the form of a three dimensional lookup table.

The search space for the range of motor speeds that were tested is shown in Table 7-3. These figures were derived from Equation 7-1, with the number of positions in the table set at 570 (10x3x19). The search space for the beam with its three dimensional array far exceeded the search space that was used for the pendulum.

speeds	search space
2	3.9×10^{171}
3	9.1×10^{271}
5	2.6×10^{398}
11	3.9×10^{593}

Table 7-3. Balancing beam lookup table search space.

Reproduction

Two point crossover was used using the x-axis (ball position) and y-axis (beam position) of the array as the positions within the array to be cut. The first cut points of the crossover were determined by randomly choosing points between 0-18 and 0-9. The end cut points of the crossover were determined by randomly choosing points between the first cut points and the end of the array, 18 or 9. The chromosomes between these two parents were swapped as shown in Figure 7-14. A mutation rate of two percent was chosen, with every individual in the population being mutated after crossover occurred.

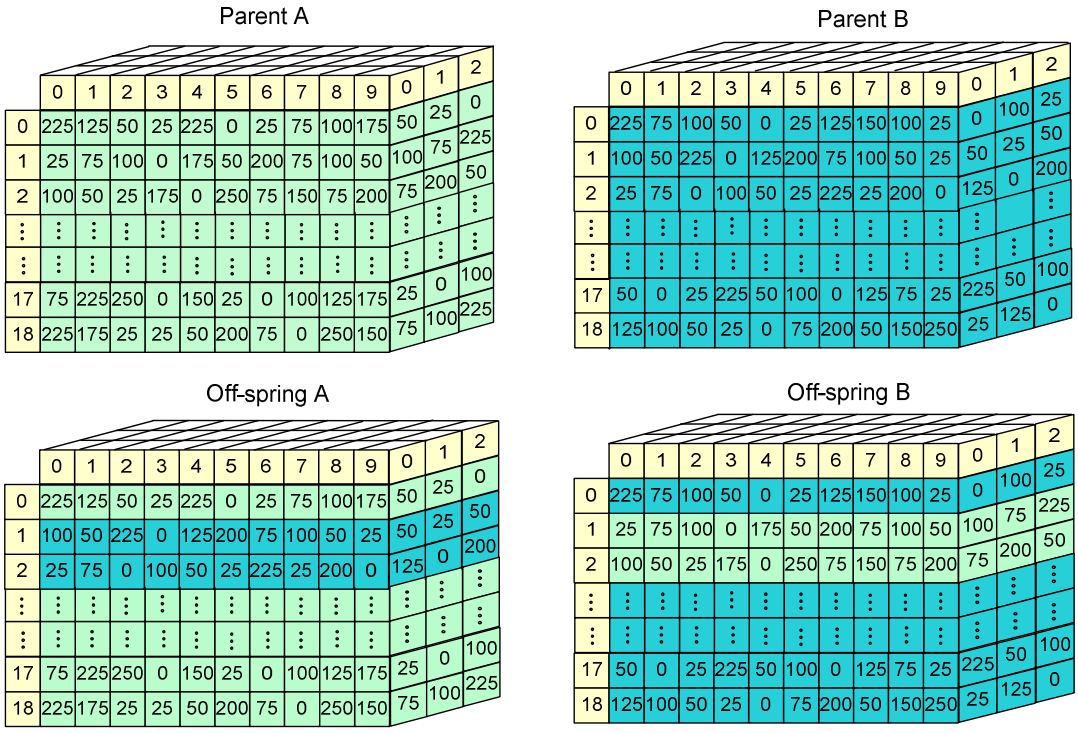


Figure 7-14. An example of reproduction of ball-balancing beam chromosome using two point crossover.

Selection

The selection process used was similar to that of the mobile inverted pendulum. The selection process was tournament with a group size of two, giving a moderately low selection pressure but maintaining a higher diversity in the population after selection. The selection process stepped through the population sequentially and compared the fitness of two adjacent parents, keeping the parent with the higher fitness. After the selection process was finished, the population was shuffled so that future selection processes acted on different groupings of parents.

7.2.3 Simulation and Coding Structure

The simulation used the equations as shown in Equation 7-2 and Equation 7-3. The derivation of these equations is described in chapter six. These equations were configured for a one millisecond time period, with a new ball position and speed calculated on each time step. Correspondingly the distance at which the beam moved was set for the same time period. The maximum beam movement was calculated from the real beam system, using two maximum motor speeds of 125 and 250 pulses per second, or a beam angular velocity of 22.7 and 45.4 degrees per second. The simulation used the motor speed and direction to calculate the new beam position. From this the new ball speed and position was calculated for the next millisecond and then fed back to the lookup table. The actual time that the ball was in motion was calculated from the number of times the simulation was called using the one millisecond time period as a reference.

$$x_{new} = x + \frac{1049v}{2^{20}} + \frac{101x + 24b}{2^{24}} \qquad \textbf{Equation 7-2}$$

$$v_{new} = v + \frac{786x + 184b}{2^{16}} \qquad \textbf{Equation 7-3}$$

The simulation kept the ball and beam parameters as a 32-bit integer number, which needed to be converted to a value the lookup table could use. These values were nineteen ball positions, ten beam positions and three ball speeds. Therefore a series of if-else statements were used to convert the simulation integer numbers to the lookup table requirements. In a similar manner, the motor speeds from the lookup table were converted into a simulation value that was added to or subtracted from, the current beam position to give a new beam position after one millisecond had passed.

Coding structure

```
main()
{
        open data files and     print number of speeds and motor speed
        do
        {
                setup();  //create a randomized population and set the time
                do
                {
                        reproduce();  //perform crossover and mutation
                        find_fitness(); // call simulation to find the fitness all the individuals
                        selection(); //perform tournament selection
                        generation++;
                        if ( !(generation %5))
                                store(); //store maximum fitness, average fitness, runtime
                        if (record_pop && (!(generation%50)))
                                record_population(); // store the population
                } while( max_fitness < 300000);  //repeat test until maximum fitness is reached
                if (record_last_run)
                        record();       //store the chromosome of the best individual and replay it
                repeat_test++;
        } while (wanted_tests != repeat_test);      // repeat evolutionary process
}
```

7.2.4 Fitness Evaluation

The individual's fitness was determined by how long the ball remained balanced on the beam before hitting either end-stop. At the start of each test, the beam was placed in the horizontal position and the ball was at rest. The simulation was then run until either the ball hit an end-stop or 60 seconds had passed. Each individual was tested seven times with the ball positioned at seven different locations on the beam, giving a combined total maximum fitness for each individual of 420 seconds.

7.2.5 Results

Initial experiments employed a two dimensional lookup table which used only the beam and ball positions. It was found that this information alone was not enough to provide a successful evolution, so the lookup table was modified to provide for a third parameter incorporating speed.

Two ranges of experiments were performed with two maximum stepper motor pulse rates. The first used 125 pulses per second which equated to a maximum beam angular velocity of 22.7 degrees per second. This was the speed of the actual beam motor and was at the limit at which the beam could control the ball. The second experiment used 250 pulses per second which equated to a maximum beam angular velocity of 45.4 degrees per second. For each experiment, four ranges of motor speeds (two, three, five and eleven speeds) were evaluated.

The first experiments used eleven start positions ranging from ±18 degrees relative to the top of the beam. Initial experiments showed that the fitness level never reached the maximum fitness. Under investigation it was found that the motor could not move the beam quickly enough to prevent those balls starting at the extreme angles from immediately hitting an end-stop. The experiment was changed to seven ball starting positions ranging from ±12 degrees relative to the top of the beam, with each individual tested seven times. A run was successful when the ball was balanced for 60 seconds, giving a combined total maximum fitness for each individual of 420 seconds.

Evolutionary stages

The graphs for these runs are presented in Figure 7-15 through to Figure 7-18. They show the relationship between the fitness of the best individual within the population and the number of generations for the four ranges of motor speeds. The graphs show step changes in the fitness level as the evolution progresses. These step changes occurred at fitness levels in the region of 180, 240, 300, and 360 seconds. These values were linked to the number of starting positions of each test and the 60 seconds that each test was performed. For every start position the beam evolved a behaviour that would bring the ball to a stable state before it reaches and end-stop. It was a simpler task to bring the ball to a stable state when the ball was started near the center of the beam, and therefore these evolved solutions were found first. The latter solutions with the ball started further from the center of the beam were harder to find, causing the fitness to plateau at these levels.

It can be seen from the graphs that the experiments with only two motor speeds evolved to successful solutions in less generations and time than the other speeds. This was partly due to the fact that the two speed chromosome required a reduced search space, as well as only using the motor at a maximum speed.

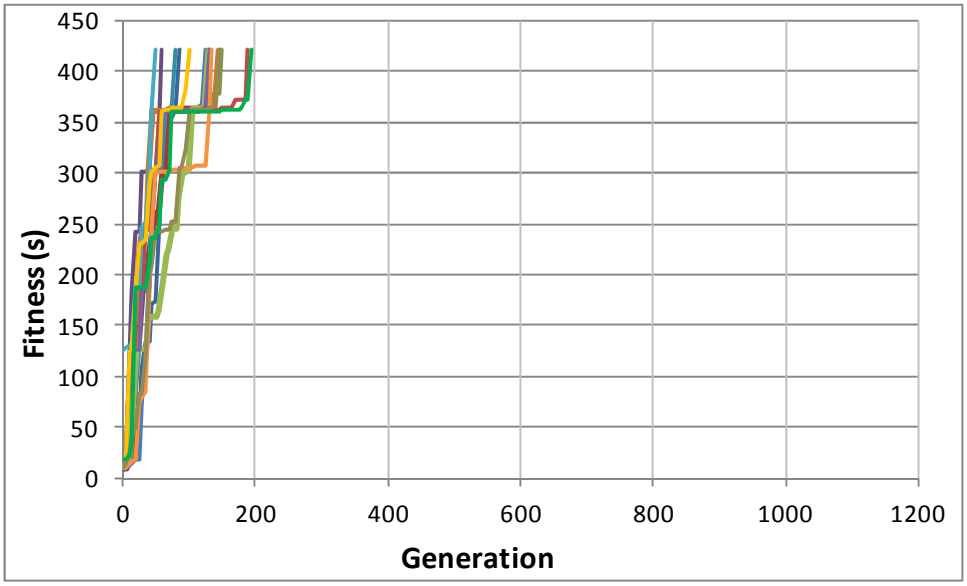


Figure 7-15. Fitness relative to generation using two motor speeds with at 8ms pulse rate showing multiple runs.

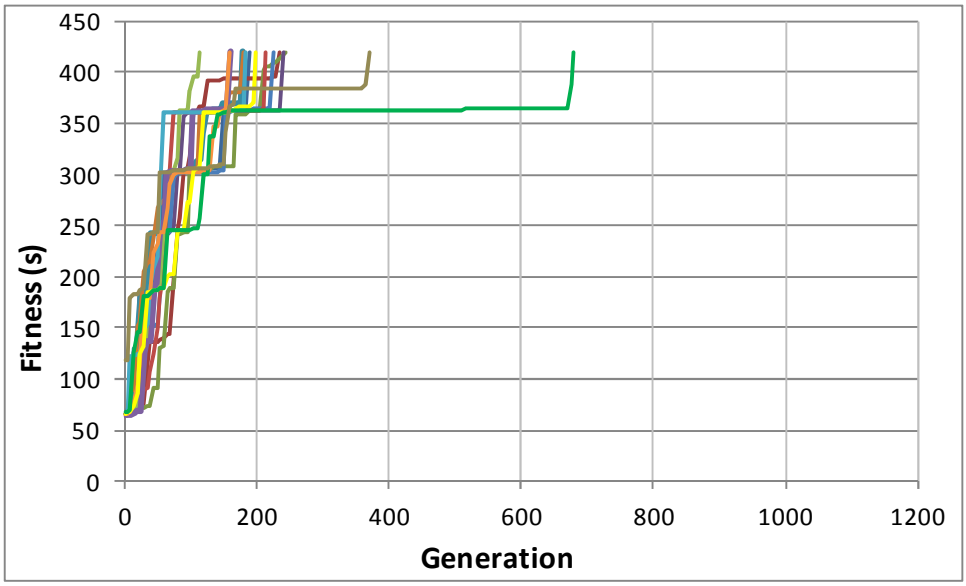


Figure 7-16. Fitness relative to generation using three motor speeds at 8ms pulse rate showing multiple runs.

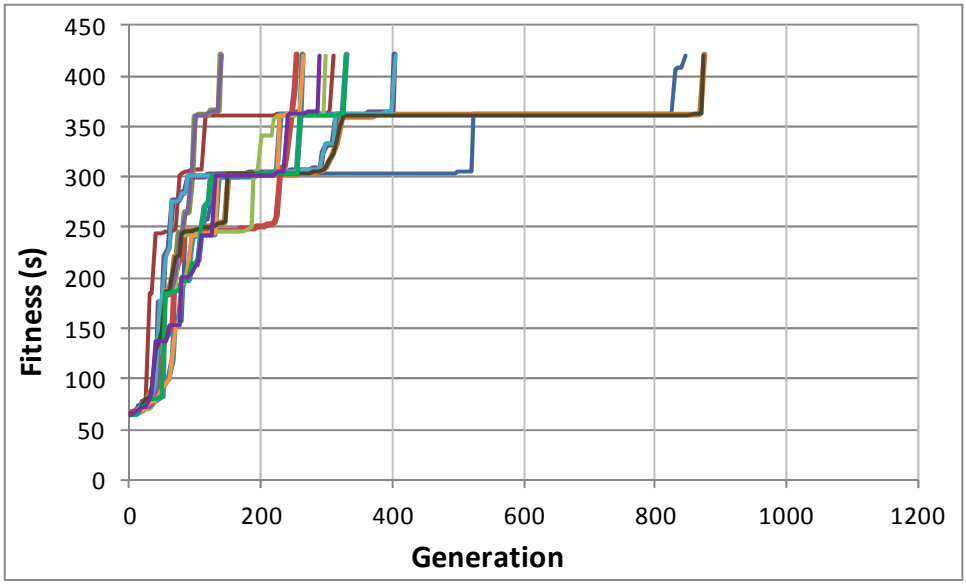


Figure 7-17. Fitness relative to generation using five motor speeds with at 8ms pulse rate showing multiple runs.

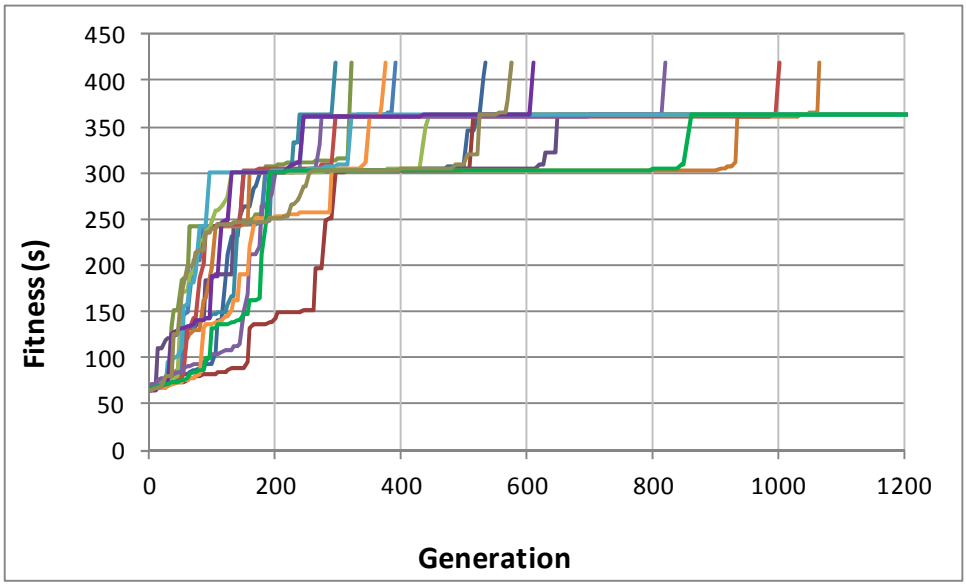


Figure 7-18. Fitness relative to generation using eleven motor speeds at 8ms pulse rate showing multiple runs

The motion of the ball and beam was observed in different stages of the evolutionary process using the graphical display. The stages were shown as:

- the ball would roll towards the beam end-stops with little or no beam motion;
- the beam would react to the ball movement, reversing the motion of the ball, however the ball would then roll to the opposite end-stop;
- the beam moved in an oscillating pattern, causing the ball to stay balanced in between two points (however after five to ten seconds the ball would break free

and gather too much speed for the beam to prevent the ball from hitting an end-stop);

- in the final stage of evolution, the beam was able to keep the ball trapped between two points for the full sixty seconds.

This characteristic oscillation of the beam was seen in all motor speed ranges. With only two speeds, the beam moved in rapid oscillations to keep the ball steady. However with a larger number of speeds, the beam would move at a slower pace. Eventually the beam evolved to keep the ball rocking between two points for all seven start positions, using an oscillating motion of the beam.

It can be seen from the graphs that there were two main plateaus in the fitness level near 320 and 360 seconds. These plateaus can be explained by the two start positions at the furthest point from the center of the beam. These were the most difficult points at which to bring the ball to a stable oscillating condition, as the ball tended to gather a high speed and was difficult to capture. This plateau was more noticeable when the experiment used five and eleven motor speeds.

For the five and eleven motor speed range, the ball would not be balanced in the middle of the beam. Instead it would be gently moved to either end of the beam and kept centered around that point. This trait can be explained by the way the ball position was determined. The position of the ball was determined by the ball sensors, and as the position of the ball is able to be determined between sensors as well as across a sensor, there are far more ball positions than the nineteen required for the lookup table. Thus the ball position is determined over a range of sensors. This range was unevenly spaced with the spacing placed closer together at the ends of the beam and further apart in the middle of the beam. This was done because it was thought that determining the ball's position and speed was more critical near the beam ends. Unintentionally however, this gave the evolved controller the best location of the ball and its speed near either end of the beam. Subsequently the evolved controller used the end locations to balance the ball. This characteristic was not seen with the two and three motor speeds experiments.

As a simulation was used, when a test was started with the ball motionless in the center of the upright beam, the evolved solution kept the motor off, so the ball stayed perfectly balanced for the duration of the test. This trait was not seen for the two speed range as the motor could not be stopped, instead the beam would move the ball to a stable position.

Evolved chromosome

An investigation of successfully evolved chromosomes and the corresponding sequence of beam and ball motions showed different patterns for each evolved chromosome. This was due to there being multiple ways of successfully balancing a ball. A successful evolution did not use a large part of the parameters in the lookup table, especially at the extreme values of beam and ball positions. The ball simply tracked to a position on the beam, and beam oscillations around that point kept it in place.

A comparison of the maximum and average fitness showed the maximum fitness increased in steps with the average fitness converging when the maximum fitness reached a plateau. At each plateau it was thought that as all the population had the same fitness, the population diversity had been lost. However an investigation of each chromosome revealed that this was not the case. This was confirmed by observation of the beam and ball motion at the plateau points. The evolution produced multiple solutions, although no individual chromosome had found a solution that would balance the ball when started in either, or both, its first and last start position. Eventually this solution was found and the evolution was completed.

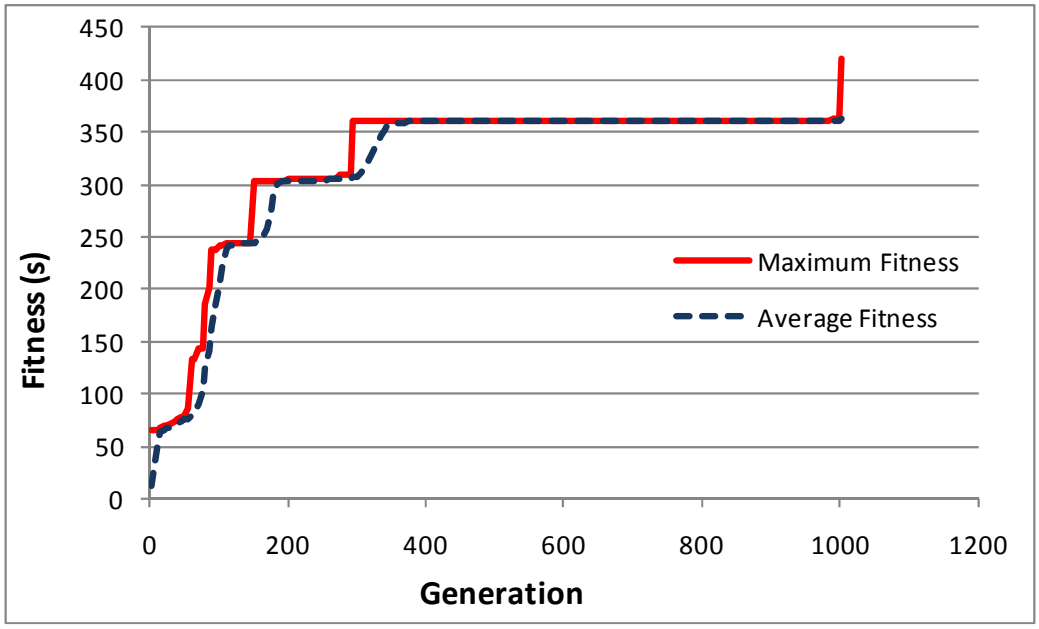


Figure 7-19. Fitness relative to generation for maximum and average fitness, with eleven motor speeds and at 8ms pulse rate.

Several hundred experiments were performed on both maximum motor pulse rates and speed ranges. Table 7-4 provides a comparison of these results showing the average fitness, number of generations, and time the evolution was in progress at the end of a

successful evolution. From this table it can be seen that the faster motor and minimum number of motor speeds had the best results in terms of the number of generations and the time taken to come to a successful evolution. It was noted that the time taken for the five and eleven motor speeds to successfully evolve was also acceptable despite the much larger search space. The reason for this was because the actual search space was reduced by: a) the ball did not travel on all places on the beam, learning to quickly come to a stable position; and b) not all the possible speeds were used, with a tendency to use the higher motor speeds.

	8ms stepper motor pulse rate			4ms stepper motor pulse rate		
speed range	generation	av fitness	time (s)	generation	av fitness	time (s)
2	118	347726	197	42	268456	35
3	268	364240	592	56	327891	76
5	398	357240	3624	98	351811	297
11	861	359427	25794	103	349563	467

Table 7-4. Comparison of the average fitness, average number of generations and the average time taken for a chromosome to evolve.

A comparison of the four motor speeds for the 8ms and 4ms maximum motor pulse rates are shown in Figure 7-20 and Figure 7-21. From these graphs it can be seen that doubling the motor pulse rate had a significant improvement on the ability of the system to evolve, especially at the five and eleven speed range. The fitness plateau at 320 and 360 seconds is clearly illustrated. All the solutions had difficulty with one or both of the extreme starting points.

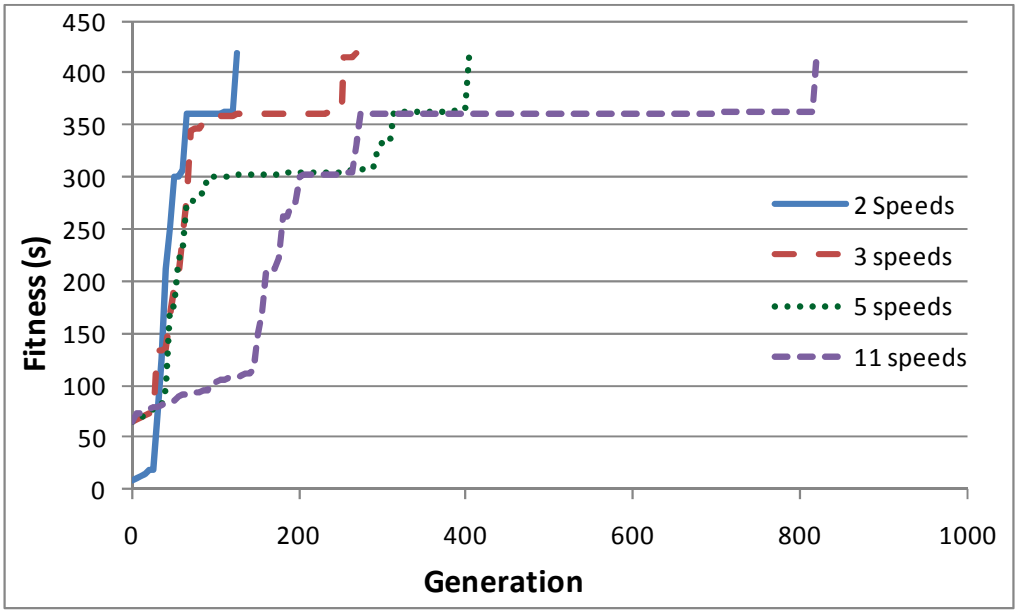


Figure 7-20. Fitness relative to generation for the four motor speeds at 8ms pulse rate.

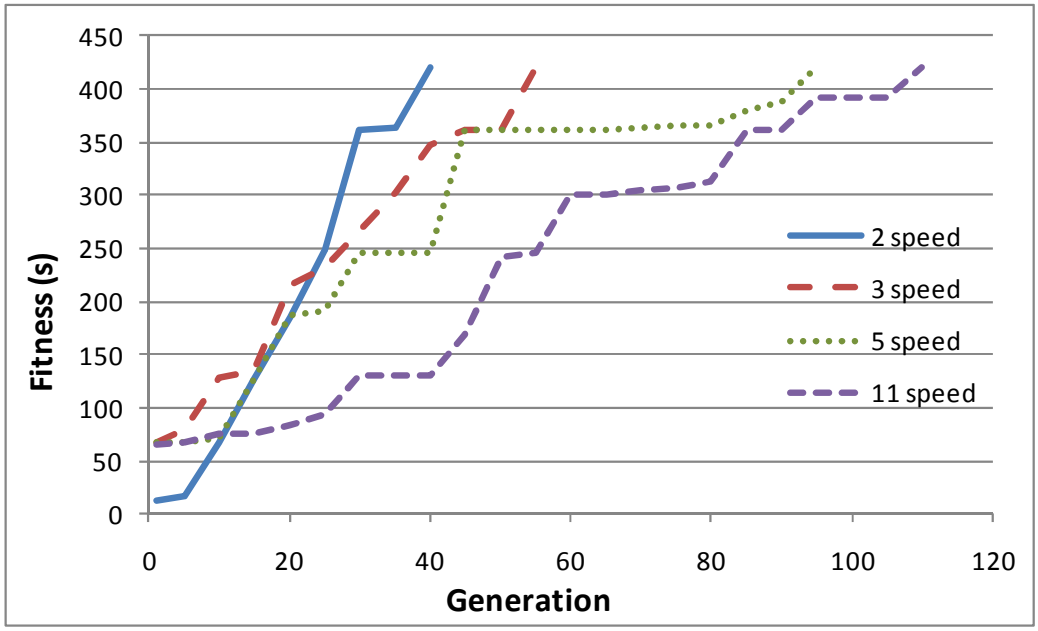


Figure 7-21. Fitness relative to generation for the four motor speeds at 4ms pulse rate.

7.2.6 Conclusions

It has been demonstrated that a robotic controller for a ball and beam system based on a three dimensional lookup table can be successfully evolved. While both motor pulse rates and all motor speed ranges were capable of being evolved to keep a ball balanced for a combined time of more than five minutes, the best evolutionary performance was achieved using a limited number of motor speeds and a higher motor pulse rate. The average time taken to evolve the circuit was dependant on the maximum speed of the motor and the number of speeds that were used. The evolution found the most difficult point of balance was when the ball was started at the angles furthest from the beam centre. This was mainly due to the slow beam motor and correspondingly slow beam angular velocity, making it difficult to stop the motion of the ball before it hit an end-stop.

A conference paper was accepted for this section on evolving a lookup table robotic controller for a ball-balancing beam with a recommendation for best paper award, and recommendation of journal publication (see chapter 1). The simulation and graphical user interface can be found in the CD accompanying this thesis.

Chapter 8

Chapter 8: Evolving a Fixed Layer Virtual FPGA for Robotic Controllers

This chapter describes how a hardware genetic algorithm was used to evolve a virtual FPGA based controller that controlled the motion of a simulated ball-balancing beam. The virtual FPGA was structured on a Cartesian architecture, with a two dimensional array of logic elements layered in four columns. The functionality of these logic elements and the routing between them was defined by a configuration bit stream. A hardware genetic algorithm was developed to evolve the virtual FPGA's configuration bit stream. It was found that after an average of 40,000 generations, the virtual FPGA could be evolved to balance the ball on the beam for more than five minutes. The simulation was modelled on a physical beam as described in chapter six and shown in Figure 8-1. The concepts of hardware genetic algorithms and virtual FPGAs have previously been discussed in chapter three.

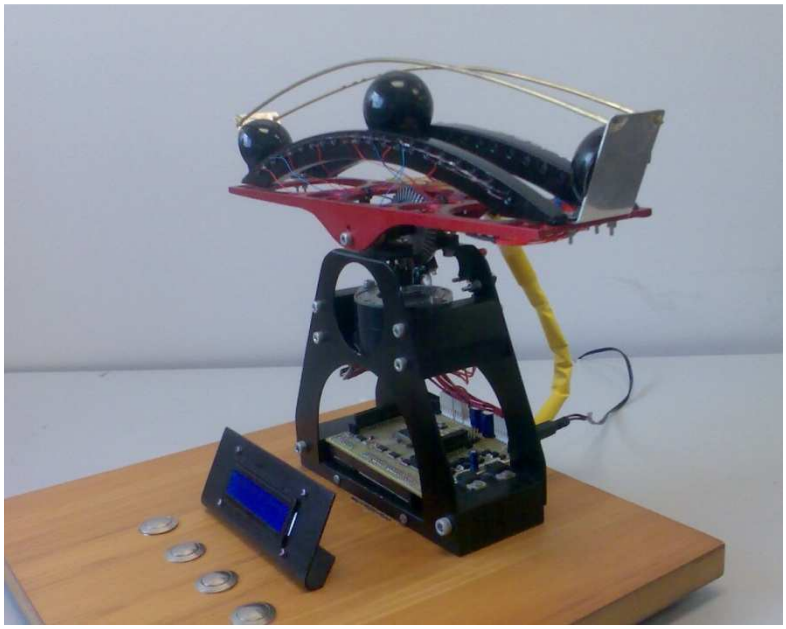


Figure 8-1. The physical balancing beam that the simulation was modelled on.

Programmable logic devices such as a FPGA are programmed with a configuration bit stream which describes the digital circuit that is to be implemented inside the FPGA. As

it is programmable, the FPGA is able to contain any digital circuit design providing there are enough FPGA resources. The FPGA is comprised of two main parts: an array of logic elements each containing a lookup table which can implement any logical function, and routing which interconnects the logic elements. Both the function of the logic element and the routing are configured by the configuration bit stream that describes the circuit to be implemented. The digital circuits within the FPGA can be evolved by a process called evolvable hardware which is a subset of evolutionary computation. In contrast to evolutionary computation where the chromosome is a possible solution, the chromosome in evolvable hardware is a possible circuit described by the configuration bit stream. This bit stream can be modified by a standard genetic algorithm and then downloaded into the FPGA. The ensuing circuit is tested for fitness, which is used in the selection process to determine which chromosomes are kept. The retained chromosomes are used to generate new offspring. This process is then repeated until a suitable result is achieved, thus the hardware itself evolves.

As explained in chapter three, there are difficulties with directly evolving the FPGA configuration bit stream. To avoid these problems a virtual FPGA was designed; this mimicked a FPGA, was suited to evolution, and could be downloaded into a normal FPGA. The FPGA contained both non-evolutionary circuits such as a processor or hardware genetic algorithm and evolutionary circuits such as the virtual FPGA. The virtual FPGA function was modified by its configuration bit stream which could be downloaded either externally from a computer via an external FPGA pin, or internally via an internal processor or hardware genetic algorithm.

The FPGA used for this experiment was the Altera cyclone EP1C12F324C8 FPGA which was incorporated on the Altium Live Design Board (refer Appendix B).

8.1 System used in Experimentation

The aim of this experiment was to evolve a robotic controller based on a virtual FPGA to balance a ball on a beam. The states of the simulated ball-balancing beam were connected to the input of the virtual FPGA. The virtual FPGA used these inputs to generate an output that was then used to control the simulated beam motor. A hardware genetic algorithm was used to modify the configuration bit stream and thus evolve the virtual FPGA. The complete system is illustrated in Figure 8-2 showing the NIOS processor, the virtual FPGA and the hardware genetic algorithm. An RS232 serial

interface was used to connect a graphical user interface to the NIOS processor allowing for control and monitoring of the evolutionary process.

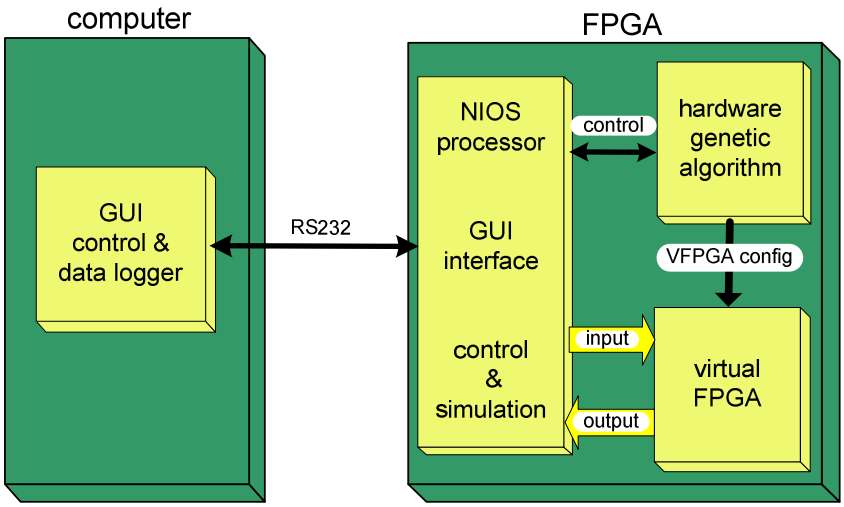


Figure 8-2. Overview of the system used to evolve the fixed layer virtual FPGA.

The system within the FPGA was comprised of three major subsystems. These were:

- NIOS processor, which provided the robotic simulation, fitness evaluation, control of the evolutionary process and data transfer of the results to the graphical user interface.
- Virtual FPGA, which was an evolutionary capable digital circuit using a four layer Cartesian based architecture with a programmable logic element at each node.
- Hardware genetic algorithm, comprised of memory storage, random number generation and chromosome mutation capabilities, which was used to evolve the configuration bit stream of the virtual FPGA.

The calculations for the simulation of the ball-balancing beam based on the model are provided in chapter six. The simulation had thirty-two outputs to match the thirty-two inputs of the virtual FPGA. These outputs were divided into nineteen ball positions, ten beam positions and three ball speeds. They were connected to the virtual FPGA as a bit sequence, in a similar manner as the actual physical beam would have supplied. For example the ball positions used sensors that were only active when the beam was broken, thus only one sensor was on at any one time. Subsequently only one bit of the nineteen ball positions provided by the simulation was active at any moment in time. This was duplicated for the ball speed and beam position bits, thus the thirty-two bit output from the virtual FPGA only had three bits active at any point in time. The virtual

FPGA would pass these thirty-two bits through its configured logic elements and routing, to provide a one bit signal to give a motor speed of either forward or backward which was sent to the simulation.

The simulation was executed on the NIOS processor, running in one millisecond time steps. On every time step it would read the direction of the motor speed from the virtual FPGA, calculate the new ball position, ball speed and beam position, and send these states back to the virtual FPGA. These steps were then repeated, allowing the virtual FPGA to control the motion of the beam and ball. The hardware genetic algorithm used only mutation to evolve the virtual FPGA configuration bit stream, utilizing the fitness derived from the NIOS processor to determine which individuals in the population were to be kept.

The interconnections between the four systems are shown in Figure 8-3. These were:

- the connection between the Quartus compiler and the Altera FPGA;
- the RS232 interface (57.6 kbps) between the NIOS processor and the graphical user interface;
- the interface between the NIOS processor and the hardware genetic algorithm;
- the link between the simulation on the NIOS and the virtual FPGA;
- the configuration bit stream sent from the hardware genetic algorithm and the virtual FPGA.

The NIOS processor ran the simulation of the balancing beam, and controlled the process of the genetic algorithm via the control lines. It could also serially read out the chromosome of the best individual using the reset chromosome counter, serial data, and serial clock lines. The hardware genetic algorithm was configured as four units to match the four layers of the virtual FPGA.

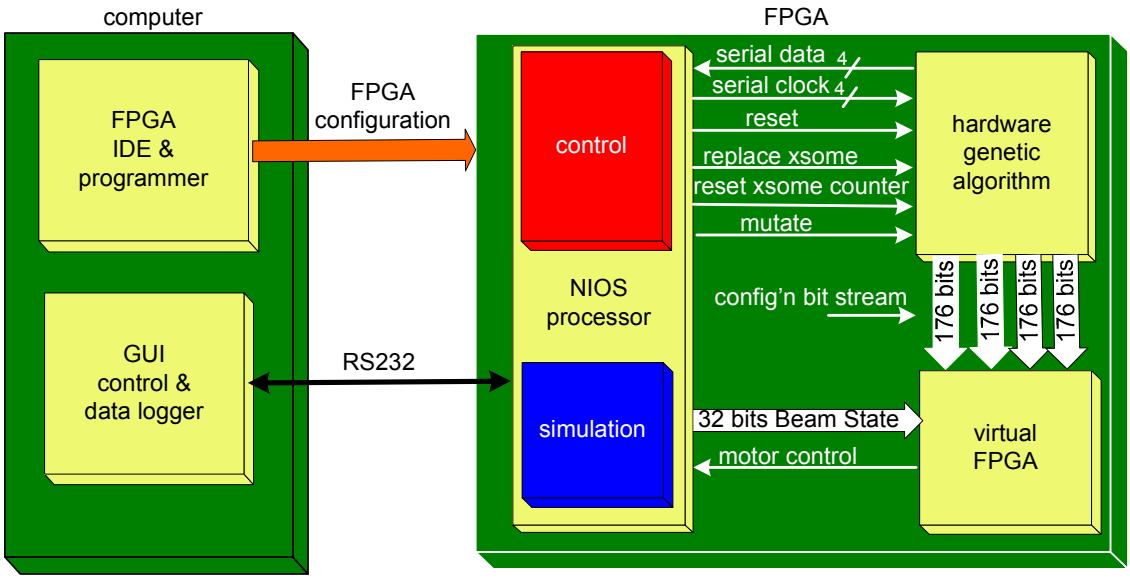


Figure 8-3. Interconnections between the three systems within the FPGA for the ball-balancing beam controller.

8.2 Fixed Layer Virtual FPGA

Initially the virtual FPGA was designed with sixteen inputs, using only the beam position and ball position from the virtual FPGA. These sixteen inputs were broken into fifteen beam positions and one input to indicate direction of ball motion (left or right). When this system was evaluated it was found that the virtual FPGA could not be evolved to balance the ball. An analysis of the results revealed that not only ball position and ball direction were required, but also that the beam position and ball speed were important components. The number of inputs to the virtual FPGA was increased. These were split into ten inputs for the beam position, nineteen inputs for the ball position and three inputs for the ball speed (indicating left, almost stopped or right). For each of the three parameters only one of the inputs was active at any one time. To accommodate the increased number of inputs, the first layer of the virtual FPGA was expanded from sixteen to thirty-two inputs.

The virtual FPGA architecture used in the experiment is shown in Figure 8-4, with thirty-two inputs and sixteen outputs. The outputs of the virtual FPGA were combined into a one bit output using an exclusive OR gate giving only two possible speeds to the beam motor: full forward or full reverse. The virtual FPGA was comprised of sixty-four logic elements which were grouped into a two dimensional array structure. This array was implemented as four layers of sixteen logic elements, with the connections moving in only the forward direction from layer one through to layer four. The layers were

connected in a feed-forward Cartesian based architecture, with each logic element within the layer able to access any two bits from the previous layer. The logic element was programmed to perform a logical operation on these two bits, and then output the one bit result to the following layer.

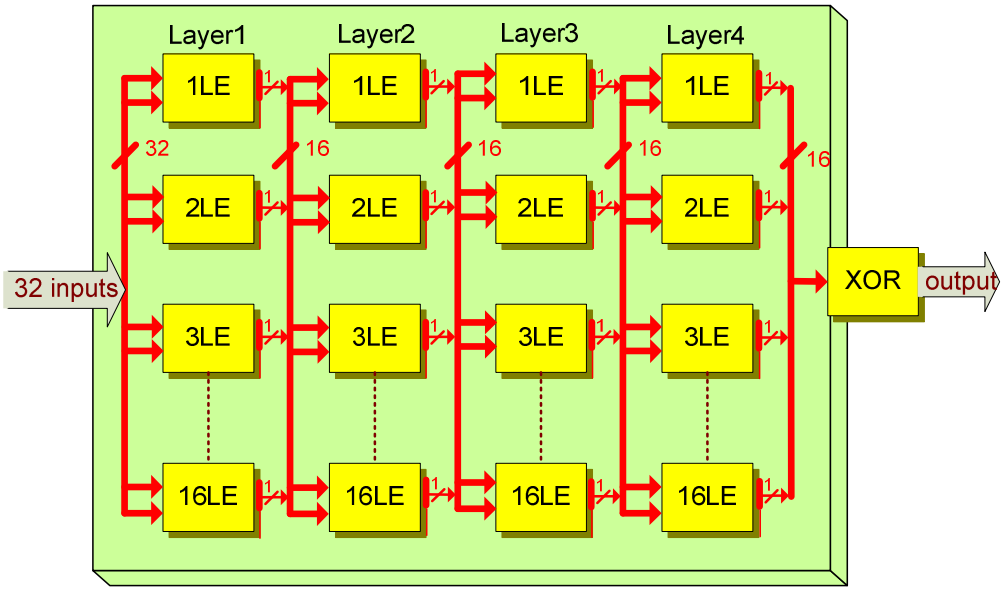


Figure 8-4. The Cartesian architecture for the fixed four layer virtual FPGA.

Logic elements

The logic elements in the second, third and fourth layers had identical architectures, however the logic element in the first layer was different to accommodate the 32 inputs. A block diagram of the logic element in the first layer is shown in Figure 8-5. The first layer had 16 logic elements each with 32 inputs that were connected to the inputs of the virtual FPGA. These 32 inputs were fed into two 1-bit multiplexers (multiplexer A and B), where each multiplexer could select any one bit from the 32 bit inputs. The 1-bit output from each multiplier was then fed into a function table which could select between two functional operators. These were select source A and source !B. The output of the logic element was 1-bit which was combined with the other 15 logic elements in the first layer to present 16 bits to the following layer (layer two). The number of configuration bits for each element was 11, 5 for each multiplexer, and 1 for the function table. With 16 elements in each layer, a total of 176 configuration bits were required for the routing and logic elements in the first layer.

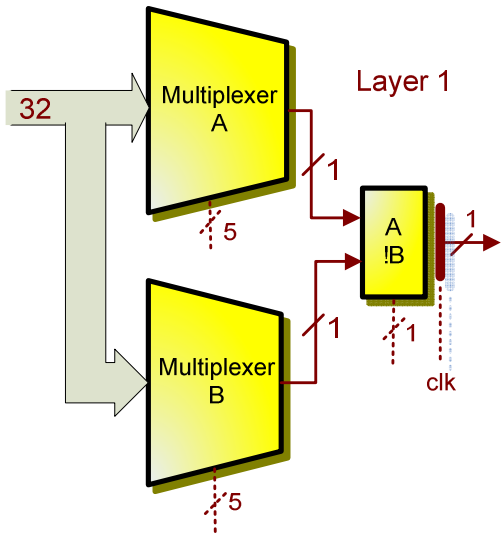


Figure 8-5. A logic element in the first layer of the fixed layer virtual FPGA.

The second, third and fourth layers each had 16 logic elements with each logic element connected to the 16 outputs from the previous column. The logic elements were the same for each of these layers, as shown in Figure 8-6. The two, one bit multiplexers could select any two bits from the previous layer and feed these into the function table.

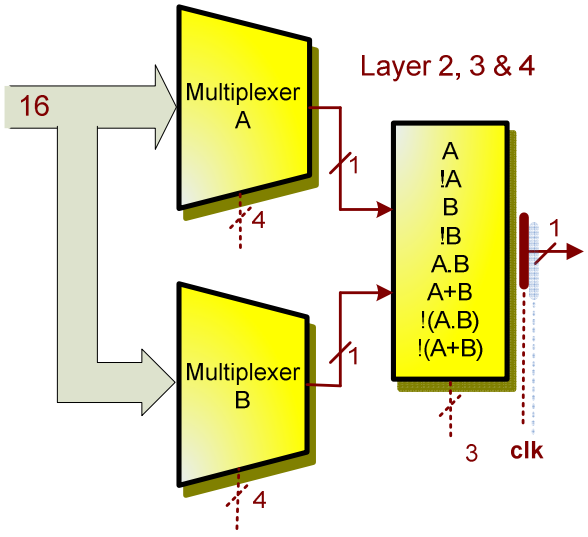


Figure 8-6. A logic element in layers two to four of virtual FPGA.

The function table had eight logic operations as shown in Table 8-1. The two inputs to the function operator were selected by the two multiplexers, the one bit output from the function operator was sent to the next layer.

selection	function
000	A
001	!A
010	B
011	!B
100	AND
101	OR
110	NAND
111	NOR

Table 8-1. List of functional operators for the fixed layer virtual FPGA.

The main difference between the first and the subsequent layers is that the first layer had 32 inputs, requiring 10 configuration bits for the multiplexers, leaving only 1 bit for the functional table, whereas the following layers had 16 inputs requiring 8 configuration bits for the multiplexers, leaving 3 bits for the function table, enabling it to have a greater range of function operators. Once again each logic element had only a 1 bit output; this was combined with the logic elements on that layer to provide a 16 bit output to the following layer. The final output from layer four was sixteen bits, which was reduced to 1 bit with the addition of an exclusive OR circuit giving two motor states, forward and backward.

The control of the multiplexers and which function was to be used in the function table was set by the configuration bit stream. A total of 11 bits were required to configure each logic element, thus with 16 elements per layer a total of 176 bits were required to configure each layer, giving a total of 704 bits for the four layers of the whole virtual FPGA. The search space of this chromosome can be derived from Equation 7-1 giving a value of 2^{704}. This equates to a search space of 8.4×10^{211}.

The configuration bit stream was kept as four separate entities (one for each layer), and it was this bit stream that was evolved by the four hardware genetic algorithms.

8.3 Hardware Genetic Algorithm

The hardware genetic algorithm was implemented as four duplicate sections working in parallel which interfaced to the configuration bits stream of the four layers of the virtual FPGA as shown in Figure 8-7. The total configuration bits required for the complete virtual FPGA were 704 bits (176 bits per layer). Two extra configuration bits were used on each layer to set the mutation rate. Each hardware genetic algorithm stored the chromosome for its associated layer and worked independently from the other layers.

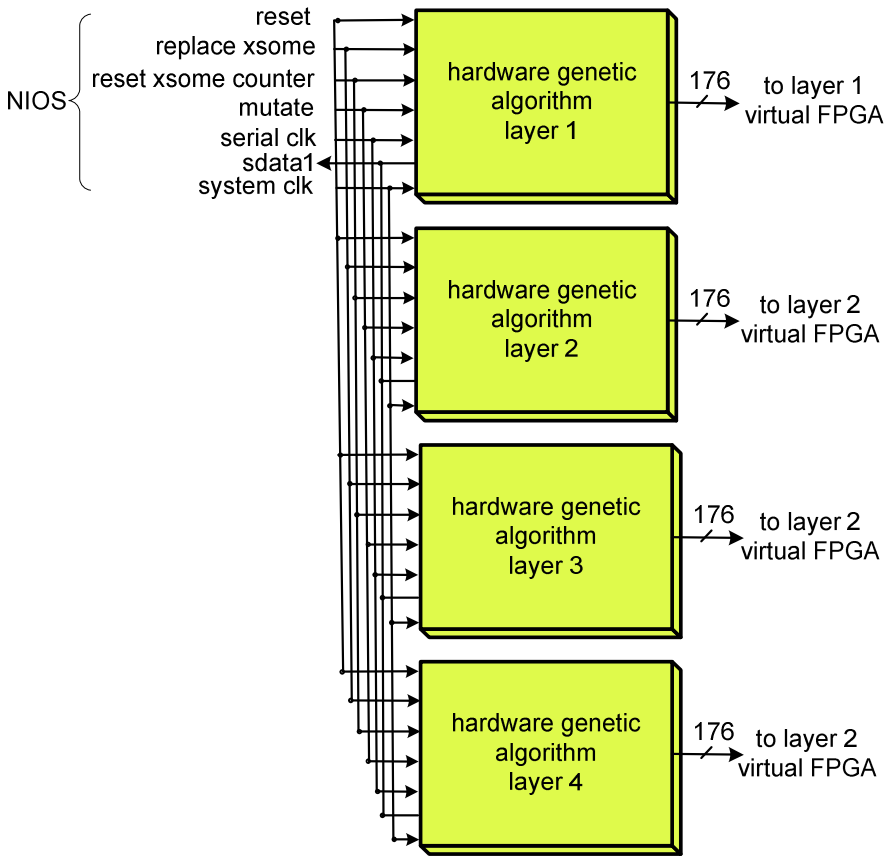


Figure 8-7. The NIOS and fixed layer virtual FPGA connections for the hardware genetic algorithm.

The operation of the hardware genetic algorithm was controlled from the NIOS processor with the aid of six control lines and one serial data output. These were

- reset, which generated a new chromosome from the random number generator;
- replace chromosome, which replaced the parent chromosome with the mutated offspring;
- mutate, which generated a new off-spring from the parent using mutation;
- system clock, which was used for the random number generator and system synchronization; and
- serial clock, serial data, and reset chromosome counter, which were used to send the best chromosome to the graphical user interface for display and storage.

The hardware genetic algorithm was based on a paper produced by Wang [89] with modifications to the population size and mutation rate. The genetic algorithm used only mutation in its reproduction of new offspring. By not implementing the crossover operator the number of logic elements required by the FPGA was reduced. This allowed the genetic algorithm to be implemented within a relatively small FPGA. The FPGA

that was used in the experiments in this research was the Altera Cyclone EP1C12F324C8 which had a capacity of 12,000 logic elements. The final circuit including the NIOS processor, virtual FPGA, and hardware genetic algorithm, used 86% (10,543 logic elements) of the Cyclone EP1C12F324C8 FPGA resources. In comparison a high end Cyclone IV has over 100,000 logic elements, and a high end Stratix V has over a 1,000,000 logic elements.

Each hardware genetic algorithm was comprised of three main sections as shown in Figure 8-8. These were a random number generator, storage for the best chromosome, and a mutation unit. The configuration bit stream was connected in parallel to the virtual FPGA. This parallelisation increased the routing resources within the FPGA, but greatly reduced the time taken to load the configuration bit stream, in comparison to a serial bit stream.

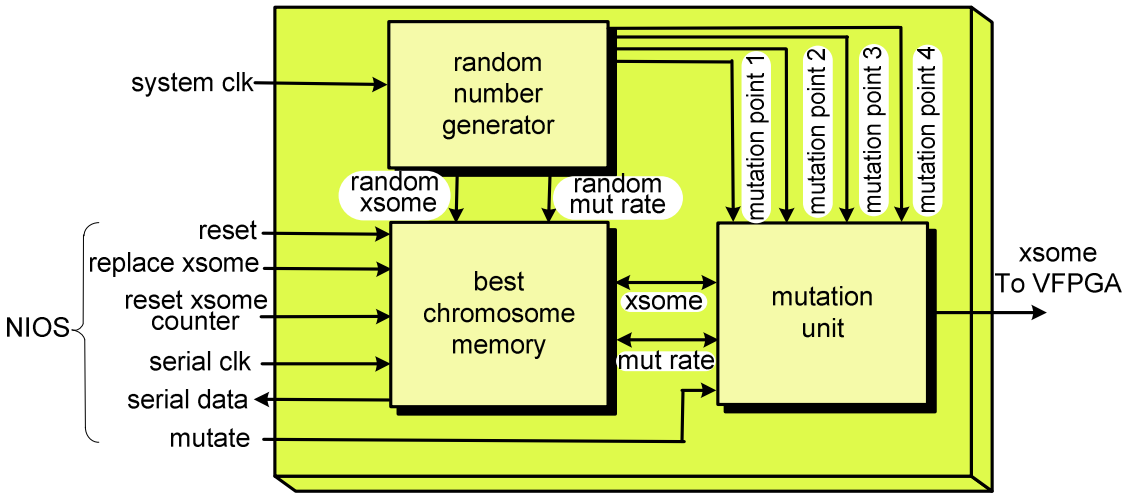


Figure 8-8. System and interconnections within the hardware genetic algorithm with an evolving mutation rate.

8.3.1 Random Number Generator

The random number generator used a linear feedback shift register to generate both the initial 178-bit starting chromosome (2 bits for mutation), and four 10-bit random mutation points that were used to mutate random bits within the configuration bit stream.

8.3.2 Memory Storage

The best chromosome memory stored the current chromosome under review. If the fitness level of the offspring chromosome generated by the mutation unit was equal or

better than the parent chromosome, then the parent chromosome would be replaced. It was found under test conditions that replacement of the individual at a fitness level equal to or above the fitness level of the parent (rather than only above), had a greater chance of evolutionary success as it allowed the individual to move more easily through the adjacent search space, reducing the possibility of stagnation at local maxima. This genetic algorithm had a lack of diversity with its limited population and therefore did not exhibit a strong selection process; however it was shown that it did work as a hardware genetic algorithm for a virtual FPGA. At any stage the best chromosome could be read back to the graphical user interface via the NIOS processor.

8.3.3 Mutation Unit

In a standard genetic algorithm the Bit Mutation Rate is the number of bits that will be varied as a percentage of the entire chromosome. It is assumed that the possibility of a mutation occurring on a chromosome is 100 percent, where a mutation will always occur on a set number of bits. In the system used in this hardware genetic algorithm there was also a Chromosome Mutation Rate. This was the probability that a mutation would occur within the chromosome. Unlike standard genetic algorithms that have a 100 percent probability of a mutation occurring, this system had a the possibility that a mutation would not occur at all.

Each mutation unit was comprised of four, for-loop structures that inverted a random bit in the configuration bit stream. The mutation bit position within the configuration bit stream was determined from the random number generator, and the number of bits within the configuration bit stream to be mutated could be varied. The mutation unit used a large amount of the FPGA resources as loop structures are highly inefficient forms of hardware.

The length of the chromosome required to configure the fixed four-layer virtual FPGA was 176 bits per layer. Two extra bits were added to the chromosome to set the rate of mutation, giving four possible mutation rates. As the mutation rate was now encoded within the chromosome, it was changed by the genetic algorithm.

Mutation rate

A hardware random number generator was used to produce four random 10 bit numbers ranging from 0 to 1023. These random numbers determined which bit in the 178 bit

chromosome would be mutated. It should be noted that as the random number exceeded the actual size of the chromosome, there was a possibility that a mutation would not occur. The four possible mutation rates, set by the two bits in the chromosome, determined whether one, two, three or four mutation points within the 178 bit chromosome would be altered. Consequently both the chromosome mutation rate and the bit mutation rate could be varied. The possibility of a chromosome actually mutating varied with the bit mutation rate. This can be calculated using Equation 8-1 and Equation 8-3.

$$m_r = \frac{xn}{r} \qquad \textbf{Equation 8-1}$$

$$m_p = 1 - m^{nl} \qquad \textbf{Equation 8-2}$$

$$m = 1 - \frac{x}{r} \qquad \textbf{Equation 8-3}$$

where

m_r is the mutation rate
x is the chromosome length
r is the length of the random number
n is the number of mutations
m_p is the mutation probability
l is the number of layers
m is the probability that a mutation will not occur

From this equation the possible chromosomes mutation rates, and bit mutation rates for the four mutation rates for the complete chromosome of 704 bits can be calculated. These are shown in Table 8-2.

mutation bits	mutation probability	maximum bit mutation rate
1	53%	0.56%
2	77%	1.12%
3	90%	1.68%
4	95%	2.24%

Table 8-2. Mutation rate settings.

8.4 Graphical User Interface

8.4.1 Overview

The graphical user interface for the ball-balancing beam controlled by a virtual FPGA, as shown in Figure 8-9, was different from the graphical user interface described in chapter seven. This was because the data used to show the ball-beam states and evolutionary process was obtained from the NIOS processor in the FPGA via an RS232 serial link. The interface could show:

- the motion of the ball and beam;
- the ball and beam states;
- the generation number and fitness;
- the control buttons for the hardware genetic algorithm and the simulation.

When the display of the motion of the beam was enabled, the NIOS processor would continuously send the ball and beam states generated by the simulation to the computer. This however severely increased the software overheads of the NIOS processor and subsequently slowed the evolutionary process; therefore the visual display was normally turned off.

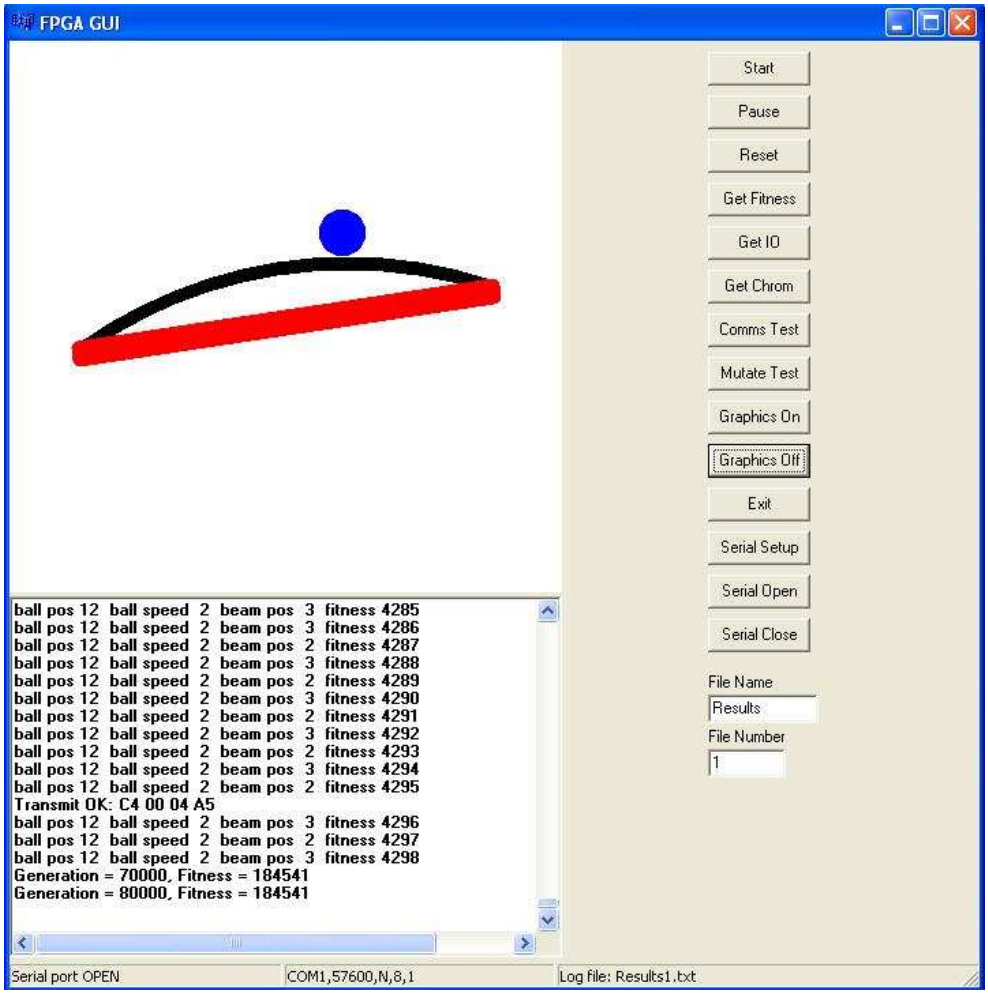


Figure 8-9. Graphical user interface used for the fixed layer virtual FPGA controlling the ball-balancing beam.

The NIOS processor was programmed to send the current fitness at regular intervals which was stored along with the current time. At the end of a successful run, the final chromosome was stored. If required, the final five minute run of ball and beam motion using the successful chromosome were replayed and stored for later analysis.

The buttons on the graphical user interface are grouped into:

- evolution control, start, pause, reset, exit;
- serial communications, serial setup, serial open, serial close;
- graphics, on, off;
- testing, testing the communication between the GUI and NIOS processor, and the mutation function;
- data request, manually reading the current fitness, uploading the chromosome or reading the current inputs and outputs of the simulation.

8.4.2 Data Communication Protocol

The communication between the graphical user interface on the computer and the NIOS processor on the FPGA was via an RS232 serial port running at 57.6 kbps. A serial communications protocol was developed so that data and control information could be passed between these two systems. An example of a transmission of four bytes of data is shown in Table 8-3 where the instruction (5) and data (10, A4. FF, E4) are sent. The header, data packet and end of packet was generated by the transmitter and sent to the receiver. The receiver waited until it received:

- the header byte C4;
- the number of bytes in the data packet;
- the instruction;
- the data packets;
- the end of packet A4.

If the end of packet was not found, or the total number of data packets was incorrect, the instruction and data packets were discarded. The same protocol was used on the return path between the NIOS processor and graphical user interface.

parameter	value
header	C4
total number of packets	4
instruction	5
data packet	10
data packet	A4
data packet	FF
data packet	E4
end of packet	A4

Table 8-3. An example of the serial transmission of four bytes of data.

The commands for the transmission of data from the graphical user interface to the NIOS processor are shown in Table 8-4. These commands are broken into control, data request and testing. The control commands required no response from the receiver, while data request and test commands did.

command	value
start genetic algorithm	0
pause genetic algorithm	1
stop genetic algorithm	2
reset genetic algorithm	3
get fitness	4
get chromosome	5
get inputs & outputs	6
graphics on	7
graphics off	8
test communications	9

Table 8-4. List of commands used for data transmission from the graphical user interface to the NIOS processor.

The commands from the transmission of data from the NIOS processor to the graphical user interface are shown in Table 8-5. The data associated with these commands were included in the packet. The genetic algorithm finished command was used to tell the computer that the desired fitness had been reached.

command	value
sending fitness	0
sending inputs and outputs	1
sending chromsome	2
sending beam state (graphics on)	3
genetic algorithm finished	4
communications test	5

Table 8-5. List of commands used for data transmission from the NIOS processor to the graphical user interface.

8.5 Simulation and Coding Structure

8.5.1 Simulation on Computer

Ideally in this experiment the simulation, fitness evaluation, and genetic algorithm would be performed on the computer rather than the NIOS processor. This was because the computer had a much faster clock speed (3 GHz as opposed to the 50 MHz on the NIOS processor), and a more powerful processor architecture with a floating point co-processor. However the simulation was executed on the NIOS processor because of the limited speed at which data could be transferred between the computer and the NIOS processor over the serial link. The only computer interface on the Altium FPGA Live Design Board used in these experiments was a RS232 serial port with a maximum data rate speed of 57.6 kbps. This link made the evolutionary process run extremely slowly

as the chromosome and ball beam states were required to be transmitted from the computer to the virtual FPGA and simulation.

If the genetic algorithm was run on the computer then it would need to send a new chromosome to the virtual FPGA for each fitness evaluation. The total number of bits to send for the complete chromosome (704 bits) is shown in Equation 8-4. With a baud rate of 57.6 kbps, it would take 16.ms to send this chromosome from the computer to the NIOS processor. If there were 100,000 chromosomes to be tested then the transmission time for these chromosomes would be 27 minutes.

$$920 = 704(chromosome) + 32(header) + 184(start/stop) \quad \textbf{Equation 8-4}$$

However this was not as time intensive as sending the simulation data. The simulation needed to send 32 bits for the beam states to the virtual FPGA, which equates to a total of 80 bits as shown in Equation 8-5.

$$80 = 32(states) + 32(header) + 16(start/stop) \quad \textbf{Equation 8-5}$$

Similarly the virtual FPGA needed to send the motor speed to the simulation, which equates to 50 bits as shown in Equation 8-6.

$$50 = 8(motor) + 32(header) + 10(start/stop) \quad \textbf{Equation 8-6}$$

Thus 130 bits are sent for each step in the simulation, equating to a transmission time of 2.26ms. The simulation used the ball-beam states and motor speed in one millisecond time steps. Assuming no time was taken for the simulation calculations then the time taken to perform a test over a simulated run of 300 seconds is 300,000 data transmissions at 2.26ms giving a time of 677 seconds. If we assume the evolutionary process required 100,000 individuals to be tested then the time taken to perform the complete evolution would be 67.7 million seconds, equating to 2.15 years. Therefore although it would seem that the computer was a far better place to run the simulation and genetic algorithm it was impractical due to the slow speed serial link between the computer and FPGA.

8.5.2 Simulation on NIOS

The simulation operated in one millisecond time steps, using the output of the virtual FPGA to move the beam in the appropriate direction. The new ball position, ball speed and beam directions were then calculated, and fed back to the virtual FPGA as a 32 bit

number. The circuits within the virtual FPGA would then produce a new motor direction depending on these new inputs. As the NIOS does not have a floating point co-processor, the ball beam states were kept as long integers. These variables were converted into a 32 bit output using an if-else structure.

Code structure

The coding structure running on the NIOS is shown below, detailing the interfaces to the hardware genetic algorithm and the onboard simulation for the fitness evaluation.

```
main (){
        setup_NIOS(); //enable interrupts for the UART receiver
        while (1){
                if (start_flag) { // set by computer to start the genetic algorithm
                        mutate_xsome();
                        generation++;
                        current_fitness = get_fitness(); // run the simulation
                        if(send_state_flag)  // can request current fitness
                                send_fitness(generation, max_fitness);
                        if(current_fitness >= max_fitness) { // replace chromosome
                                max_fitness = current_fitness;
                                replace_xsome();
                        }
                        //note computer can request to see dynamic display of ball-beam by
                        //setting send_state_flag
                        if (max_fitness  reached) {
                                start_flag = 0; //stop the simulation
                                send_fitness(generation, max_fitness); // send results
                                send_xsome(); send the entire chromosome
                                send_state_flag = 1;
                                get_fitness();          //send the final beam run.
                                send_state_flag = 0;
                                send_data_to_PC(instruct_new_test,0); // say running new test
                                reset_xsome();
                                generation = 0;
                                max_fitness = get_fitness();
                                send_fitness(generation, max_fitness);
                        }
                        else if(!(generation % 250)) //periodically send generation and fitness
                                send_fitness(generation, max_fitness);
                }
        }
```

```
}
```

Instructions were sent to the NIOS from the computer via the serial port, and were activated in an interrupt function. When a new command was received, the instruction was stored and the new_instruction flag was set. The instruction would be executed as illustrated in the action_command function shown below.

```
void action_command(void) {
        switch (instruction) {
                case test_comms:  //echo instruction back to the PC
                        send_data_to_PC(test_GA, total_bytes);  break;
                case start_GA:
                        start_flag = 1; break;
                 case pause_GA:
                        start_flag = 0; break;
                case get_IO_GA:
                        send_IO(); break;
                case send_state_on:
                        send_state_flag = 1;  break;
                case send_state_off:
                        send_state_flag = 0; break;
                case stop_evol_GA:
                        start_flag = 0;
                        send_fitness(generation, max_fitness);
                        send_xsome();
                        send_data_to_PC(instruct_stop,0);
                         reset_xsome();   break;
                case get_xsome_GA: //read the current xsome and send to the PC
                        send_xsome(); break;
                case get_fit_gen_GA: //read the current xsome and send to the PC
                        send_fitness(generation, max_fitness);  break;
                case reset_GA:
                        reset_xsome(); //reset GA, gen random xsome, & send to the VFPGA
                        generation = 0;
                        max_fitness = get_fitness(); //evaluate a new fitness for xsome
                        send_fitness(generation, max_fitness);
                        send_xsome();
                break;
        }
        new_instruction = 0;
}
```

8.6 Fitness Evaluation

The beam was evaluated in only one start position as shown in Figure 8-10, with the ball positioned at the far left side of the beam, and the beam tilted towards the right, at an angle of twenty degrees from the horizontal plane. At this angle the ball would naturally move towards the centre of the beam without any movement of the beam itself. The fitness was determined by the time taken before the ball hit either end-stop; this was the length of time that the beam could balance the ball.

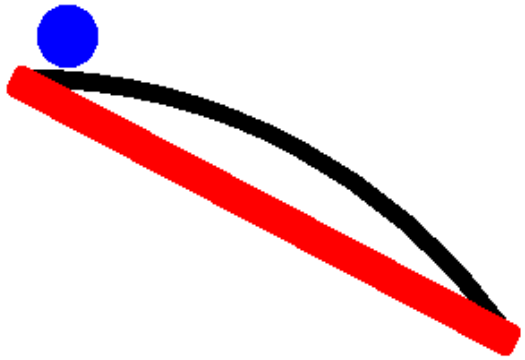


Figure 8-10. Starting position of ball on the beam.

8.7 Results

Once a command for a test run was sent by the graphical user interface on the computer, the NIOS processor automatically controlled the evolutionary process interacting with the hardware genetic algorithm without further commands from the computer. The NIOS processor was programmed to automatically send the generation number and the current fitness to the computer every thousand generations. The computer could also request the current fitness and current generation at any time. As well the computer could request to continuously receive the ball and beam status allowing the motion of the ball and beam to be seen dynamically in real time. This feature was normally disabled as it severely slowed down the simulation and thus the time taken for genetic process to complete. The beam and ball positions could also be recorded for later analysis.

The evolution was set to end when the ball hit an end-stop or when the fitness had reached 500 seconds. At the end of the genetic process the final chromosome was stored and the last simulation of the successful chromosome repeated, with a step by step recording of the beam position, ball position and ball speed sent to the computer. This

allowed the motion of the ball in relation to the beam, to be monitored. Two motor speeds were evaluated, one set to pulse every one millisecond (giving a beam rotation of 181.6 degrees per second), the other two milliseconds (giving a beam rotation of 90.8 degrees per second).

A typical fitness level for the start of the experiment would be approximately 0 or 600 milliseconds. An investigation of the motion of the beam at this stage revealed that the beam would either not move, or move in the wrong direction. If there was no beam movement, the ball would roll down the slope and after 600 milliseconds hit the right end-stop. If the beam motion was in the wrong direction, the ball would roll to the left and immediately hit the left end-stop. After several generations this behavior would change and a movement of the ball to the right would cause the beam to tilt left, changing the ball direction from right to left. The ball would then hit the left end-stop. As the evolution progressed, the speed of the returning ball was reduced allowing the ball to change direction several times; however the ball would not go into a stable state, and would eventually gain too much momentum for the beam to correct its motion before it reached an end-stop. These unstable ball motion patterns would have a fitness level ranging from 10 to 50 seconds.

Eventually the fitness would jump from this plateau to a successful result. When these individuals were analysed, it was found that the ball would spend most of its time near one end of the beam, moving between two points in close proximity. This caused the beam to rock backwards and forwards trapping the ball in a semi stable state. Eventually the ball would gain enough momentum to break away from this position and move towards the opposite end of the beam. The beam would then respond by bringing the ball back to its semi stable state where it would repeat the process. As this pattern was repeated, rapid improvements in the fitness were achieved.

A graph of the fitness versus generation for a motor pulse rate of one millisecond is shown in Figure 8-11. It can be seen that the number of generations taken to evolve to 500 second fitness for this motor speed ranged from 18,000 to 52,000 generations with an average generation of 32,000.

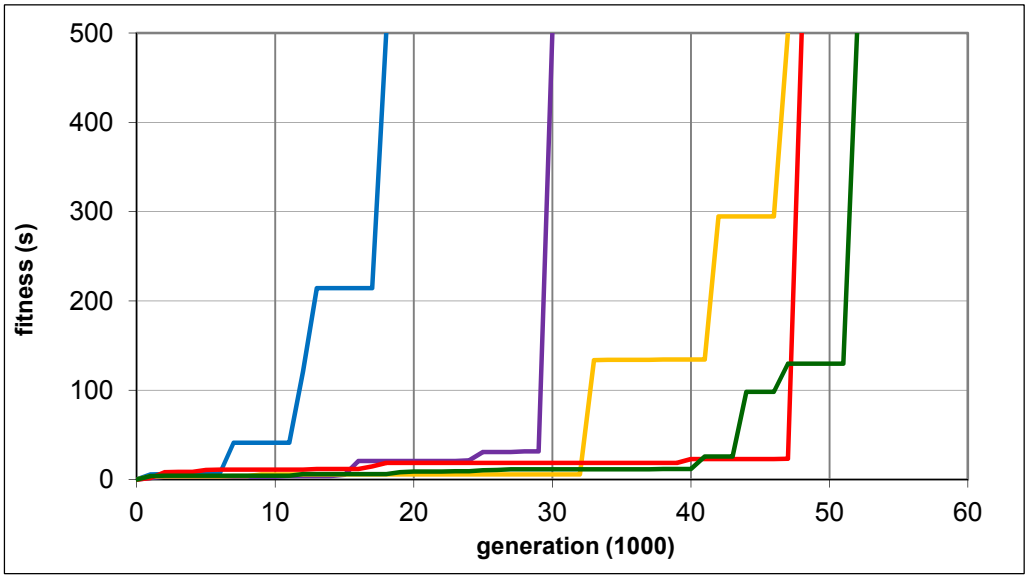


Figure 8-11. Fitness relative to generation for a 1ms motor pulse rate.

The graph of the fitness versus generation for the motor pulse rate of two milliseconds is shown in Figure 8-12. It can be seen that the number of generations taken to evolve to 500 second fitness for this motor speed ranged from 41,000 to 330,000, with an average value of 240,000 generations.

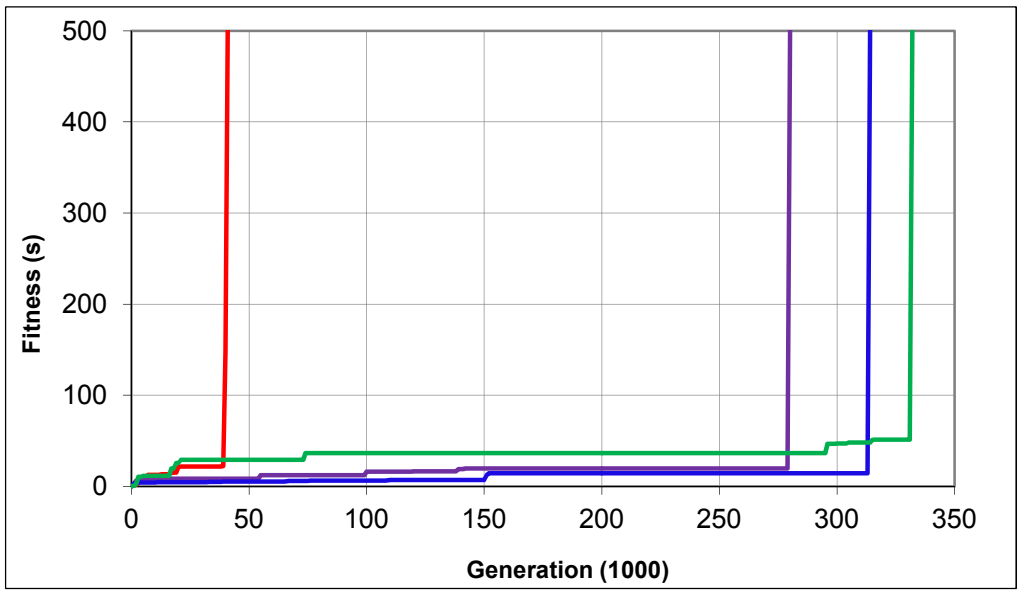


Figure 8-12. Fitness relative to generation for a 2ms motor pulse rate.

A recording of the ball and beam jittering motion which the beam used to capture the ball in a stable position was recorded and plotted on the graph shown in Figure 8-13. It can be seen that the beam is swinging backwards and forwards around two points, which moves the ball in alternate directions, keeping it in a relatively stable position. This tended to be towards one side of the beam, however eventually the ball would break out of this pattern and move towards the opposite side of the beam. At a set point

the beam would correct for this and bring the ball back to the two original set points, allowing the pattern to repeat indefinitely.

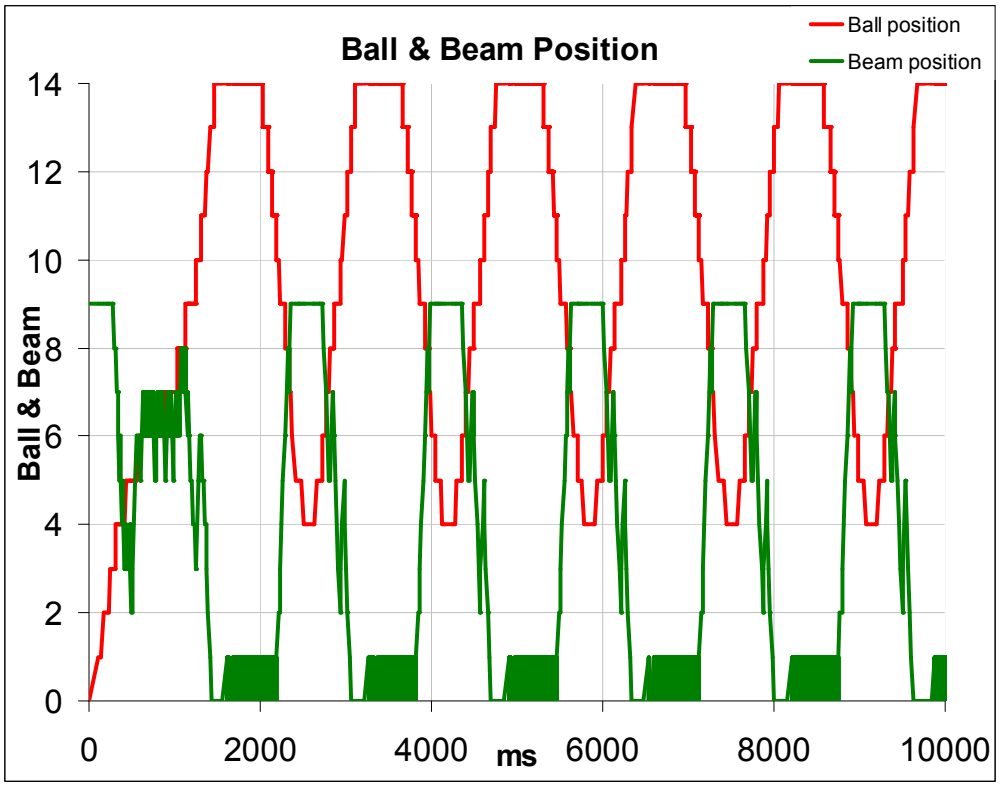


Figure 8-13. The motion of the ball and beam showing the oscillating pattern which is keeping the ball in a stable position.

8.8 Conclusion

This chapter has shown how a virtual FPGA acting as a robotic controller can be evolved to balance a simulated ball-beam system. A commercially available board with an FPGA was used in the experiments to contain a digital electronic circuit in the form of a virtual FPGA, a hardware genetic algorithm, and a beam simulation running on a NIOS processor. The simulation provided the current ball position, beam position and ball speed to the virtual FPGA, while the virtual FPGA provided the appropriate motor direction back to the simulation.

The evolved virtual FPGA could balance the ball on the beam for more than 500 seconds. An analysis of the ball's trajectory during a successful run showed the ball oscillated between two closely spaced sensor positions in a semi-stable state. When the ball moved beyond these points, the beam reacted in a self correcting manner to bring the ball back to its semi-stable state.

For a motor speed of 181.6 degrees per second it was found that on average, less than 40,000 generations were required to evolve a circuit able to balance the beam for more than five minutes. The faster motor speed required fewer generations to evolve a suitable fitness; however both motor speeds performed satisfactorily.

A conference paper was produced for this section on evolving electronic circuits for robotic control (see chapter 1). The Verilog code, C code and graphical user interface can be found in the CD accompanying this thesis.

Chapter 9

Chapter 9: Using Hardware Simulation for Evolving Robotic Controllers

This chapter investigates the implementation of a genetic algorithm using a hardware simulation rather than a software simulation, for the evaluation of each chromosome's fitness within the population. The chromosome describes the control system for the robot, detailing how the robot will react to events within the robotic environment. A simulation is required by the genetic algorithm to model the actions of the robot and its environment in order to evaluate how well each chromosome performs as a controller. Typically the simulation is written in software and executed sequentially on a processor. However if the simulation could be written in a hardware description language, then it could be implemented as a digital circuit within a FPGA, giving a significant improvement in speed. To test this proposal, two identical genetic algorithms (except for the simulation) were developed, one using a software simulation, the other a hardware simulation. These two systems were implemented and a comparison performed.

The full simulation of the beam used in chapter eight was modified to a simpler model with the removal of code that was used to refine the position of the ball. This simulation was generated in both software and hardware and a measurement of the characteristic of both simulations was performed to ensure they were similar in nature. The simulations were then evaluated on identical genetic algorithms so that a valid comparison could be performed between the two simulations.

This chapter also presents a more advanced virtual FPGA architecture than that used in chapter eight, incorporating more powerful functions within the lookup table, and a reducing layer architecture requiring fewer configuration bits and therefore a smaller search space than the flat layer architecture.

In the first experiments it was found that the hardware simulation evolved a successful digital circuit in a time period that was approximately seventy times faster than the

software simulation. In this experiment the hardware simulation's speed was limited to 5MHz, matching the speed limitation of the virtual FPGA with its internal clock delays. A second experiment was performed with the delays removed from the virtual FPGA allowing the hardware simulation to operate at 50MHz. This enabled the hardware simulation to run at a speed approximately 700 times faster than the software simulation.

This chapter explains how the simulation and fitness calculations were shifted from software to hardware with their associated control systems. It details the systems used in the genetic algorithm process, explaining their operation and control, and describes the reducing layer virtual FPGA architecture. Finally the results are presented with comparison between the hardware and software simulation.

9.1 Overview

A genetic algorithm is an iterative process that repeats three tasks: reproduction, fitness evaluation, and selection. The time taken for a population to evolve is split between these three processes, with the reproduction and selection taking comparatively little time compared to the fitness evaluation. This is because all the individuals within the population must be evaluated on a simulation of the robot and its environment to determine each individual's fitness. This fitness is used by the selection process to select which individuals will be retained for the next generation. The time taken for the simulation to test each individual increases as the evolution progresses, due to the increasing average fitness of the population. The simulation for a robotic controller is a computer model of the robot and its interactions with its environment. This simulation may include floating point and trigonometry calculations. If the simulation process can be sped up, then a large improvement in the time taken for the evolutionary process to produce a successful individual will result.

The software coding for a simulation algorithm using floating point arithmetic and trigonometric calculations on a computer is relatively straight forward, as the computer has a floating point co-processor specifically designed to perform these calculations. However because of the computer's sequential coding and processor hardware structure, each calculation must be executed sequentially. If the calculations within the simulation could be performed in hardware, then many of the calculations could be performed concurrently resulting in an improvement in speed.

Software simulation

The block diagram of the system used to evaluate the software simulation is shown in Figure 9-1, illustrating how the software simulation was contained within the NIOS processor. The simulation interfaced to the virtual FPGA via the input and output lines of the NIOS processor. The virtual FPGA and genetic algorithm were implemented in hardware. In previous experiments within this thesis a full robotic simulation of the beam was used; in this experiment the simulation was modified to a simpler model using integers. Note the NIOS processor used a 50MHz clock, with the hardware multiplier and divider enabled. It should be noted that the software simulation was not executed on the computer with its higher clock speed and more powerful processor; as the amount of time required to transfer the simulation parameters for each step of the simulation between the computer and virtual FPGA is prohibitive. This is explained in more detail in chapter eight.

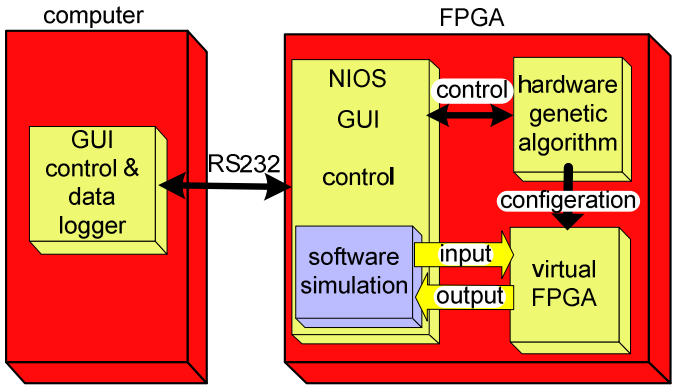


Figure 9-1. Block diagram of the systems used in the software simulation for the balancing beam.

Hardware simulation

The block diagram of the hardware simulation is shown in Figure 9-2. The simulation is constructed in hardware with the simulation's mathematical functions and fitness evaluation embedded within it. In this case the NIOS processor is only used for interfacing to the computer graphical user interface and reading the status of the genetic algorithm.

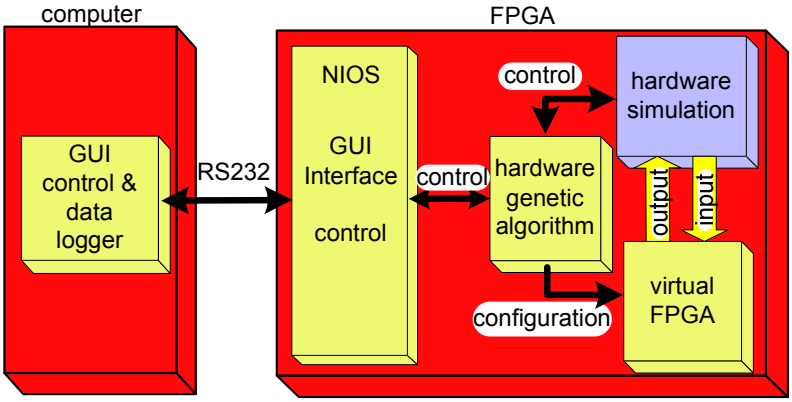


Figure 9-2. Block diagram of the systems used in the hardware simulation for the balancing beam.

System block diagram

The complete system employed for the hardware simulation shown in Figure 9-3 contained five blocks:

- the computer, used for system commands and data logging;
- the NIOS processor, used for interfacing to the computer and control of the systems within the FPGA;
- the hardware genetic algorithm, which contained the genetic operators necessary for the evolutionary process;
- the virtual FPGA, used to control the motion of the beam;
- the hardware simulation, which modelled the characteristics of the beam and ball. These blocks are described in the next section.

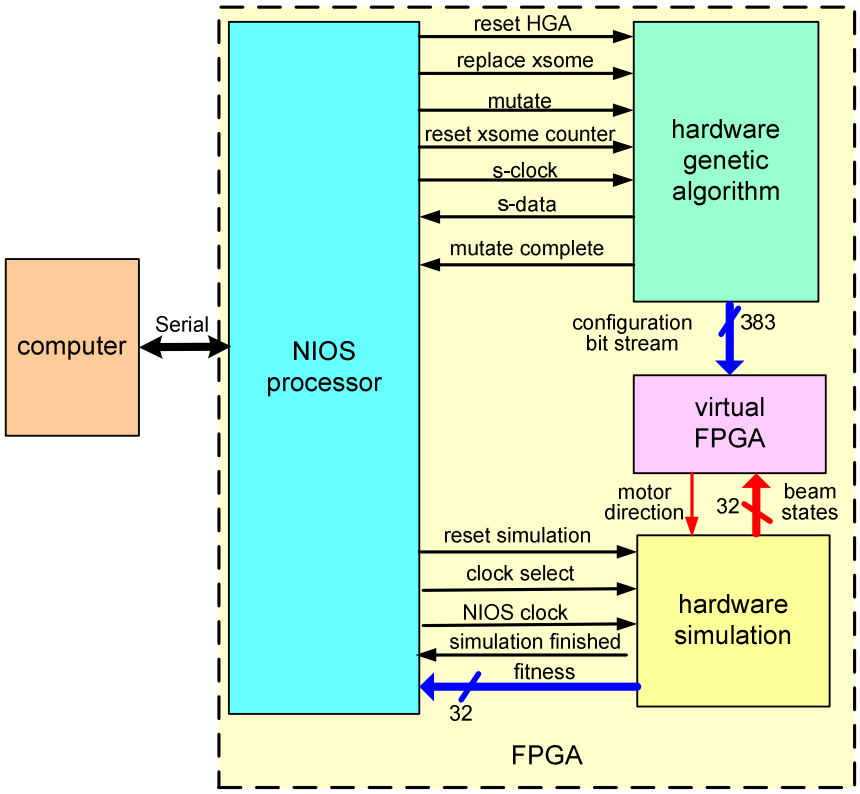


Figure 9-3. System control and data lines for the hardware simulation.

9.2 Hardware Simulation

9.2.1 Creating a Hardware Simulation

The main difficulty with creating a hardware simulation inside a FPGA is that unlike a computer, there is no arithmetic logic unit. All arithmetic formula written in a hardware description language will generate individual circuits to implement the arithmetic function, thus every occurrence of an arithmetic operator such as addition, multiplication or division, will be expressed as a complex digital circuit. In the case of floating point operations, a large amount of the FPGA logic element resources are required for each calculation due to the complexity of dealing with variables containing sign, mantissa and exponent parts. As there are typically many floating point calculations in a simulation, it becomes impractical to use this technique.

An alternative to floating point calculations is the use of integer arithmetic within the FPGA, which reduces the logic element resources required to implement the circuit within the FPGA. An integer arithmetic operation requires less FGPA resources than a floating point calculation, and thus the logic resources required for each arithmetic operation of the simulation within the FPGA is significantly reduced. Trigonometric

functions will also need to be implemented as an arithmetic approximation or a lookup table.

The first task to create a hardware simulation was to transform the floating point simulation model into an integer model. This process was straight forward although precision was lost, making the integer model less accurate than the floating point model. The algorithms needed to be checked to make sure that no arithmetic overflow occurred as the numbers were confined to 32 bits, which is a value between $\pm 2 \times 10^9$. It was also important to make sure that the timing between the arithmetic calculations and other systems was correct. The hardware simulation then had to be integrated to the virtual FPGA and the genetic algorithm.

To implement the simulation in hardware, the integer arithmetic calculations can be directly coded in Verilog (a hardware description language) using the standard multiply (*) and divide (/) syntax. The IEEE standard for Verilog 1995 allowed the use of multiplication and division for unsigned variables. This was altered by the IEEE standard for Verilog 2001 which allowed the use of signed multiplication and division within the language. The standard also introduced the ability to use signed registers (note the default value of an integer is a signed 32 bit number). Page forty-five of the Verilog 2001 standard [160] shows how signed division and multiplication can be performed. Examples of the multiplication and division Verilog code and the resultant register transfer level generated by Quartus are shown in Figure 9-4 and Figure 9-5.

```
module divide_int   (input wire signed [31:0] switch1, switch2,
                     output reg signed [31:0] result);

          always @ (*)
                    result = switch2 * switch1;
endmodule
```

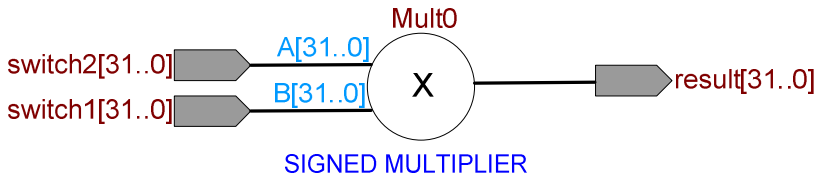


Figure 9-4. Verilog code and register transfer level description for a thirty-two bit signed multiplier.

```
module divide_int  (input wire signed [31:0] switch1, switch2,
                    output reg signed [31:0] result);

        always @ (*)
                result = switch2 / switch1;
endmodule
```

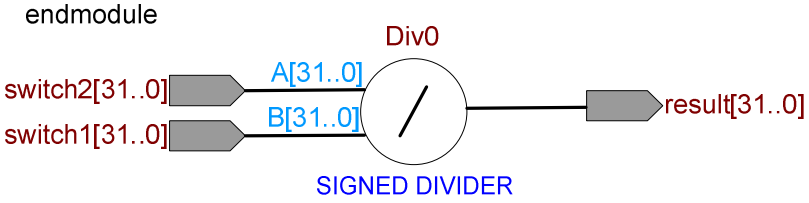


Figure 9-5. Verilog code and register transfer level description for a thirty-two bit signed divider.

The mathematical equations for the integer simulation shown in Equation 9-1 and Equation 9-2 were derived experimentally using the sensors to measure the ball position as it fell (as detailed in chapter 6). From these equations the acceleration of the ball could be found. The equations used for both the software and hardware simulation were in integer form which reduced the accuracy of the calculations but allowed the use of high-speed simulations in both software and hardware applications. The integer simulation was run on two systems, software and hardware.

$$x_{new} = x + \frac{1049v}{2^{20}} + \frac{101x + 24b}{2^{24}} \qquad \textbf{Equation 9-1}$$

$$v_{new} = v + \frac{786x + 184b}{2^{16}} \qquad \textbf{Equation 9-2}$$

Note the values chosen for the divisor were carefully chosen to equate to a number which was a power of 2. This enabled the Quartus synthesis to divide using left shifting or other minimization techniques, rather than a divide function. The RTL viewer in Quartus allowed the user to see a schematic of the hardware simulation that was generated by Quartus. It was interesting to note that the circuit was comprised of multiplexers, comparators and five signed multipliers; no dividers were used.

A comparison of the resource usage within the FPGA for integer and floating point calculations was performed. The entire simulation implemented as integers used 15% (1,900 logic elements) of the Cyclone EP1C12F324C8 FPGA resources. In comparison, one floating point calculation took 9% (1,140 logic elements) of the FPGA resources. Note Quartus does not support the IEEE 2001 Verilog real numbering system, thus floating point calculations must be performed using Quartus megafunctions.

9.2.2 Hardware Simulation Blocks

The hardware simulation, shown in Figure 9-6, was comprised of four units; the mathematical simulation unit, the fitness calculation unit, the simulation complete unit and the clock speed unit.

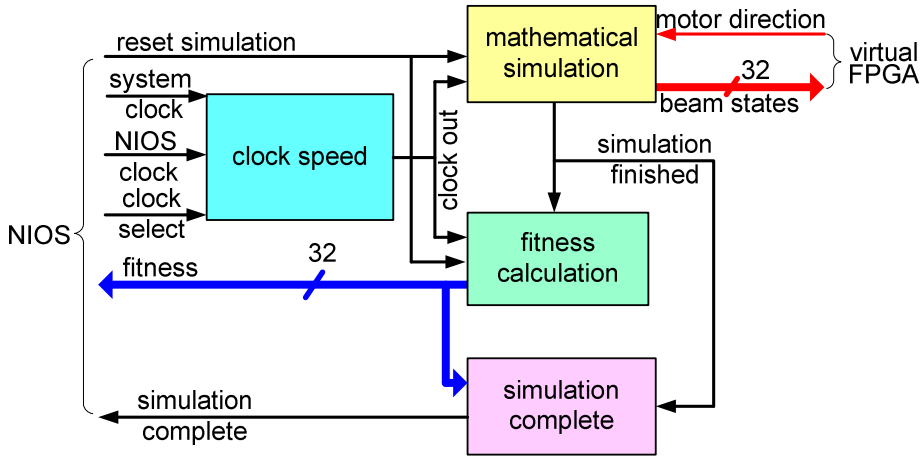


Figure 9-6. Control lines and subsystem interconnections for the hardware simulation unit.

Mathematical simulation unit

The mathematical simulation unit contained the simulation's mathematical equations implemented in hardware, with control lines connected to the NIOS processor, and the simulation input-output connected to the virtual FPGA.

The control lines were:

- reset simulation, used to reset the simulation's ball and beam parameters to the start state, and to clear the fitness level;
- clock, used to trigger the simulation. All the simulation calculations would execute simultaneously on every clock pulse, which was equivalent to one millisecond in real time;
- beam states, there were thirty-two bit outputs describing the new ball speed, ball position and beam position derived from the simulation. These were fed to the inputs of the virtual FPGA;
- motor direction, the new motor direction from the virtual FPGA, resulting from the previous beam states that had been fed into the virtual FPGA.

When the simulation reset was activated, the parameters inside the simulation were reset, placing the ball on the left side of the beam with the beam set to an angle of twenty degrees to the left above the horizontal plane. When not reset, the simulation would operate in the following manner:

- on each clock pulse the simulation would read the motor direction input from the virtual FPGA and correspondingly shift the integer value of the beam to the left or right one motor step;
- the new integer values for the ball speed and ball position were calculated;
- these new values along with the new beam position were converted into a thirty two bit binary format representing the new ball-beam state and passed to the inputs of the virtual FPGA.

The mathematical equations for the simulation were designed to calculate a new ball and beam state every millisecond. This meant that every clock pulse that triggered the simulation was equivalent to a one millisecond time period within the simulation. Thus a 5MHz clock pulse would give a simulation speed 5000 times faster than the real time event. The simulation had an output control line to show when the simulation had finished. This was set when the individual had failed the test and the ball position reached either of the two beam end-stops.

Clock speed unit

The clock speed unit allowed the simulation to run at high speed from a 5MHz clock or at a slow speed from a clock driven by a control line from the NIOS processor. This slow clock speed allowed the graphical user interface to run in graphical mode, where after each simulation step, the beam and ball states could be read and sent back to the graphical user interface running on the computer so that the motion of the ball and beam could be displayed on the computer screen. It also enabled the simulation to be paused at any stage. A clock select line from the NIOS processor was used to switch the clock from low speed to high speed. The lines used to control the operation of the clock speed unit were:

- clock select, a control line from the NIOS processor to switch between the high speed system clock or the low speed NIOS controlled clock;
- system clock, initially at 5MHz, then increased to 50MHz;
- NIOS clock, software generated clock from the NIOS processor;

- clock out, the clock source fed for the simulation and fitness unit.

Fitness calculation unit

The fitness calculation unit measured the time that the simulation ran before the individual failed. It did this by counting the number of clock pulses fed into the simulation during the evaluation of the current individual, with each clock pulse equivalent to one millisecond in real time. The fitness unit had a thirty two bit counter that was incremented on every simulation clock pulse, provided the simulation complete line was clear. The fitness counter could be read by the NIOS processor at any time, with the value of the fitness counter being the time in milliseconds that the ball had stayed balanced. The simulation finished line was also connected to the NIOS processor so that the fitness counter could be read at the end of a simulation. The lines used to control the operation of the fitness unit were:

- reset simulation, which set the fitness counter to zero;
- clock, the same clock as fed the simulation which the fitness unit counted;
- fitness, a thirty-two bit value indicating the fitness of the individual under test.

Simulation complete unit

This unit told the NIOS when the evaluation of the individual had finished, either when the individual failed, or when the fitness counter had reached 300 seconds indicating that the individual under test had obtained its maximum vale. The lines used to control the operation of the simulation complete unit were:

- simulation finished, sent from the mathematical simulation unit whenever the ball had reached an end-stop;
- fitness, used to trigger when the fitness had reached 300 seconds;
- simulation complete, this signalled to the NIOS that the simulation had finished.

9.2.3 Timing

A comparison of the time taken for the hardware and software simulation to complete one iteration of the mathematical equations in the simulation was performed.

Timing for software simulation

The time taken for the software simulation to calculate the new ball position, ball speed, and beam position, and output these as a 32 bit binary format to the virtual FPGA was measured. This measurement was taken by toggling an output pin of the FPGA at the start and end of each simulation execution as displayed in Figure 9-7. The measured time for the execution of the simulation was 16us. This time period would be periodically increased when the selection and reproduction process occurred at the end of a fitness evaluation.

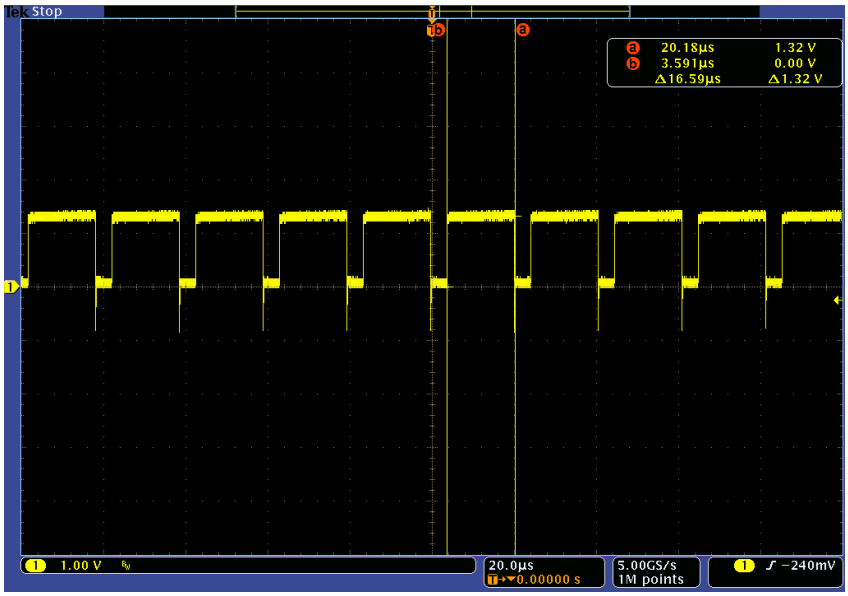


Figure 9-7. Timing diagram of the software simulation execution time.

Timing for the hardware simulation

The Altium live design board used in these experiments included a 50MHz clock which was used as a system clock for the NIOS processor and other hardware systems. This clock was also used within the virtual FPGA with each layer's output held in its past state until a clock pulse occurred. As there were five layers within the virtual FPGA, five clock cycles were required before a change in the inputs moved from the first layer through to the final layer of the virtual FPGA. This equates to a delay time within the virtual FPGA of 100ns. To match this, the clock rate chosen for the hardware simulation was 5 MHz, which equates to a hardware simulation execution time of 200ns. This enabled the virtual FPGA to respond to new values on its inputs within one execution of the hardware simulation equations.

From these measurements and calculations it can be seen that the software simulation was running at a 16us rate, whereas the hardware simulation was running at 200nS. This would give an expected speed increase of the hardware over the software simulation of approximately 80 times. This theoretical increase in speed may be less in practice due to other overheads in the genetic algorithm such as selection and reproduction.

9.2.4 Maximum Beam Step Calculations

On experimentation with the physical beam, the stepper motor required 270 pulses to move the beam from one end of its range to the other. The full angular range of the beam's motion was 60 degrees, thus one pulse from the motor gave a beam angle change of 0.22 degrees. The maximum pulse rate that the motor could operate at was eight milliseconds thus the time taken for the beam to move from one extreme to the other was 270x8ms = 2.16 seconds.

The integer value for the change in beam position after a pulse from the motor can be calculated in the following way. The maximum number of pulses to move the beam from full left to full right is 270. In order to have an integer number with a suitable resolution this value is multiplied by 10,000 to give a maximum beam movement of 2,700,000. This gave the beam a range of ± 1,350,000, with one pulse equating to an integer value of 10,000. The maximum pulse rate of the motor is 8ms, however the simulation is operating in one millisecond time steps, thus the change in the integer value after a motor pulse will be 1250 units. A fragment of the Verilog code for the beam motion is shown below. (Note the beam's maximum tilt angle has been limited as under physical experimentation on the beam it was found that once the beam went outside these limits the motion of the ball could not be controlled.)

```
if (motor_direction) begin
        if(beamPos < 900_000)
                beamPos <= beamPos + 1_250;
end
else begin
        if(beamPos > -900_000)
                beamPos <= beamPos - 1_250;
end
```

The physical beam had twenty one ball position sensors; however the virtual FPGA input constraints meant that the ball position had to be reduced to nineteen positions. It

was decided that two positions towards the middle of the beam would be removed as the information that they provided was less important than those in the centre and extremes of the beam. An if-else structure within the Verilog code as shown below was used to encode the integer value of the ball position to a binary format output.

```
if( ballPos < -185_000)
        ball_pos = 19'b1000000000000000000; //18
else if( ballPos < -165_000)
        ball_pos = 19'b0100000000000000000; //17
else if( ballPos < -140_000)
        ball_pos = 19'b0010000000000000000; //16
```

9.3 Reducing Layer Virtual FPGA

The fixed layer virtual FPGA employed for the experiments described in chapter eight was modified to a reducing layer virtual FPGA architecture as shown in Figure 9-8 with logic elements grouped into five layers. These five layers each had a reducing number of logic elements. The virtual FPGA had 32 one-bit inputs and 1 three-bit output giving 8 possible output states which were used to describe the required motor speed and motor direction. Note in this experiment only two motor speeds forward and backward were used. The number of logic elements in each layer progressively reduced, starting from 16 logic elements in the first layer and dropping to 1 logic element in the final layer. Each logic element output had 3-bits grouped together. These groups were combined with the other logic element outputs within that layer and then fed onto the following layer. It should be noted that the original grouping of the 3-bits selected in layer one were retained as they were passed through each subsequent layer.

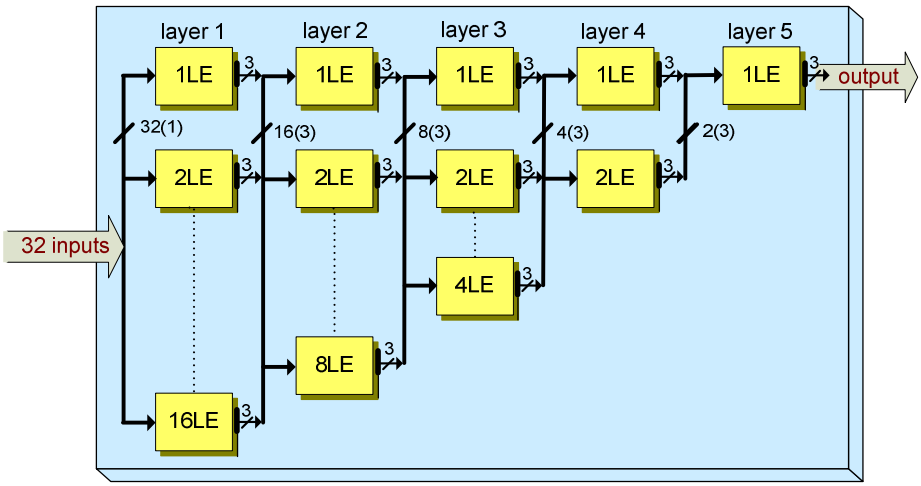


Figure 9-8. Architecture of the reducing layer virtual FPGA.

Layer one

The logic element in the first layer as shown in Figure 9-9 contained 3 multiplexers that were used to select which 3 bits of the 32 bit inputs were to be grouped together and fed onto the following layer. Each logic element in this layer had a 3-bit output which was clocked to provide synchronisation between logic elements. The grouped outputs of each of the 16 logic elements within the first layer were then fed onto layer two. Each single bit multiplexer required 5 bits to switch between the 32 inputs, and each logic element had 3 multiplexers, thus each logic element required 15 configuration bits. The total configuration bit stream for all the 16 logic elements in layer one was 240 bits.

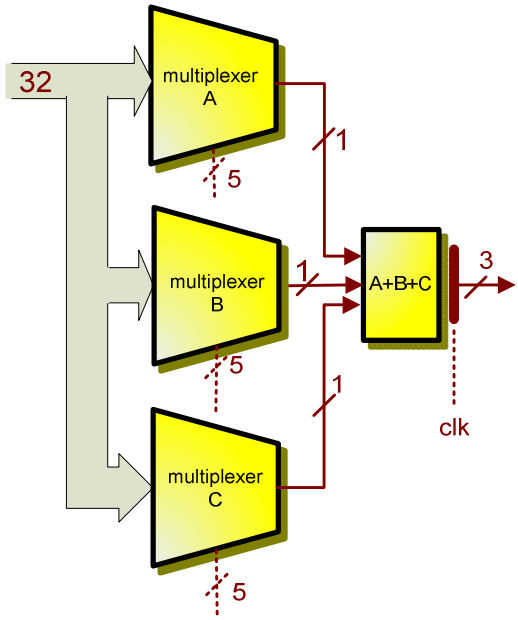


Figure 9-9. The logic element in layer one of the reducing architecture.

Layer two

The logic elements for layers two through to layer five are similar apart from the size of the input bus which progressively became smaller through the layers. The logic element in layer two, as shown in Figure 9-10, was comprised of 2 multiplexers and a function table. The 2 multiplexers each selected one of the 16 groups of 3-bits from the previous layer and fed these inputs onto the function table (groups A and B). The 3-bit groups were from a sequential grouping of the inputs to that layer, for example group one with bits 0 to 2, and group two with bits 3 to 5, through to group sixteen with bits 45 to 47. The second layer had sixteen grouped inputs, thus each multiplexer required a 4-bit selection input and the function table required 3-bits giving a configuration bit stream of 11-bits per logic element. There were 8 logic elements in this layer, thus a total of 88

configuration bits were required for layer two. The output from each logic element was one group of 3-bits with eight groups from the complete layer.

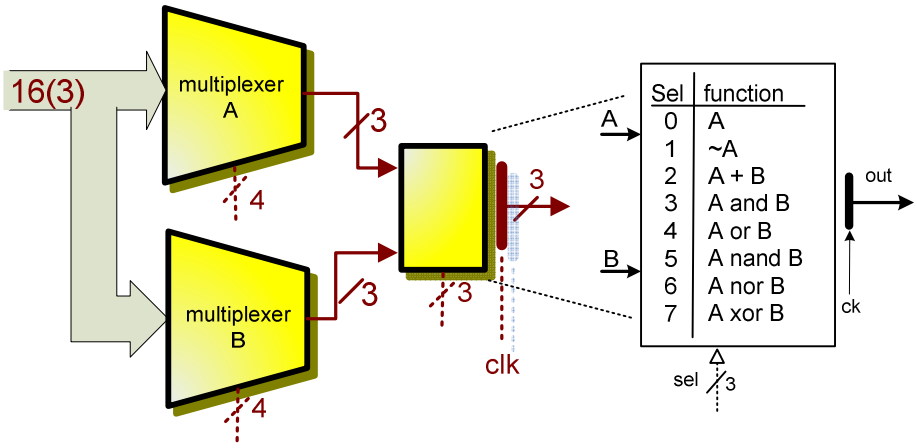


Figure 9-10. The logic element in layer two of the reducing architecture.

The function table provided simple logic functions as listed in Table 9-1. Boolean functions were performed on the two input groups of 3-bits to produce one 3-bit grouped output which was then fed to the next layer.

Select	Function	Description
000	A	only A is selected
001	~A	bitwise one's complement of A taken
010	A + B	A and B are added together
011	A and B	A and B are bitwise ANDed
100	A or B	A and B are bitwise ORed
101	A nand B	A and B are bitwise ANDed then one's complement taken
110	A nor B	A and B are bitwise ORed then one's complement taken
111	A xor B	A and B are bitwise Exclusive ORed

Table 9-1. List of functional operators for the reducing layer virtual FPGA.

Layer three

An example of the logic elements in layer three is shown in Figure 9-11. The logic element had eight grouped inputs and one grouped output, with the multiplexing process and function table similar to layer two. However because the number of inputs to the logic element had reduced, only 3-bits per multiplexer were required to select one of the eight 3-bit groupings. The number of configuration bit streams for each logic element was 9, and with four logic elements in the layer the total number of configuration bits was 36 bits for layer three. There were four grouped outputs for the complete layer.

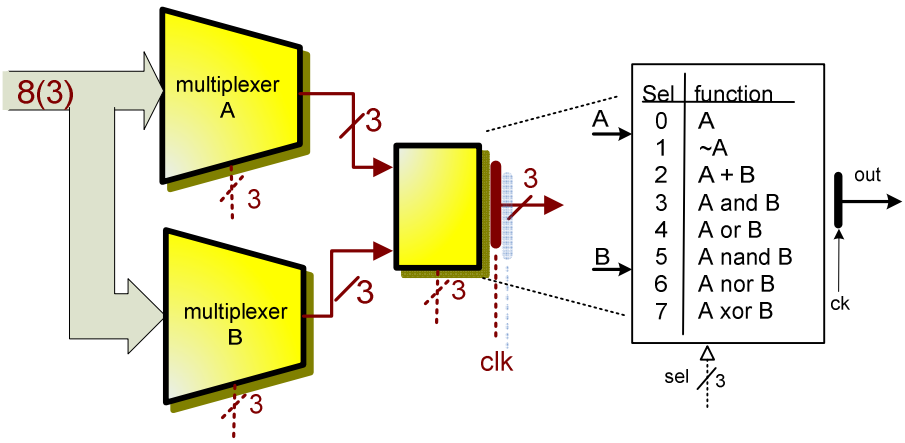


Figure 9-11. The logic element in layer three of the reducing architecture.

Layer four

One of the logic elements in layer four is shown in Figure 9-12. This is similar to the previous layer's logic elements, except that the inputs had dropped to four groups of 3-bits. Once again fewer bits were required in the multiplexers, thus the configuration bits required per logic element was now 7, and with only 2 elements in this layer, the total configuration bits for layer four was 14.

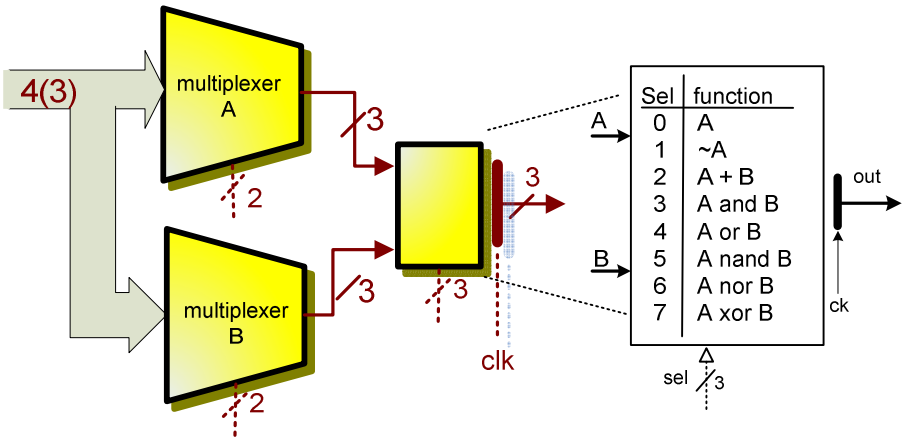


Figure 9-12. The logic element in layer four of the reducing architecture.

Layer five

The final layer had only one logic element as shown in Figure 9-13. This layer performed Boolean logic on the remaining two groups, and then output the result as a 3-bit number. The total configuration bit stream for this layer was 5-bits.

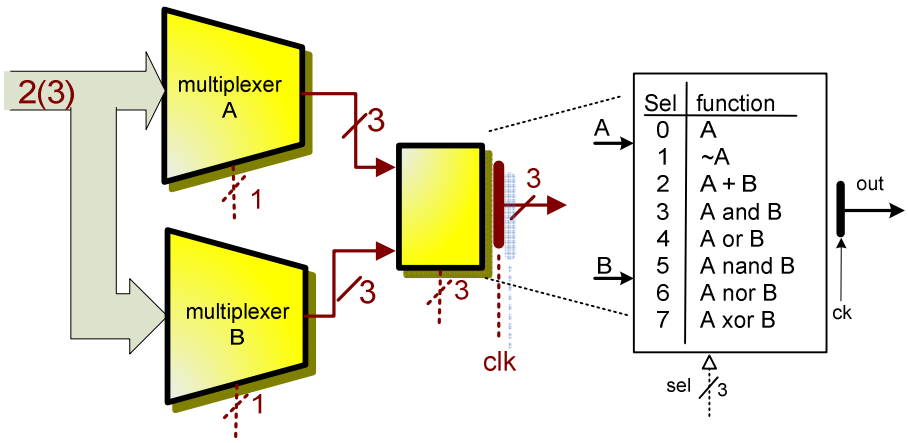


Figure 9-13. The logic element in layer one of the reducing architecture.

Combining all the configuration bit streams for the complete virtual FPGA resulted in a total number of 383 bits. This was slightly over half of what was required in the fixed layer virtual FPGA. The search space for this chromosome was 2^{383} which equates to 3.3×10^{63} as compared to the fixed layer architecture search space of 8.4×10^{211}. The advantage of this architecture compared to the fixed layer architecture was a reduction in the configuration bit length and thus a reduction in the search space leading to a reduction in the evolution time. In addition, this architecture had more powerful functions, used a three bit result to give a more complex answer, and grouped the inputs

9.4 Hardware genetic algorithm

The hardware genetic algorithm used in this experiment as shown in Figure 9-14 was similar to that used in chapter eight except: only one unit was used; it was configured for a chromosome length of 383 bits; and the mutation rates were determined by the current fitness level rather than encoded in the chromosome. The control lines between the NIOS processor and the hardware genetic algorithm were:

- reset the genetic algorithm, which generated a new chromosome and cleared the fitness;
- replace the chromosome, if the offspring was better than the parents then replace the parent with the offspring;
- mutate, which inverted one or more bits of the chromosome;
- reset the chromosome bit counter, which cleared the chromosome bit counter before reading the chromosome into the NIOS processor;

- S-clock and S-data, used to serially read the chromosome into the NIOS processor, which was then encoded and sent to the computer;

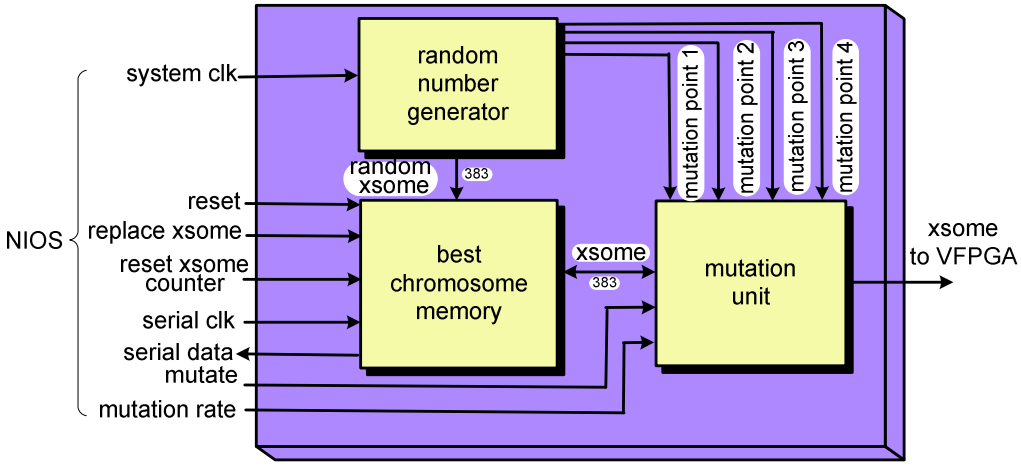


Figure 9-14. Block diagram of the subsystems within the hardware genetic algorithm.

After the evaluation of the individual was completed by the simulation, the NIOS would request the mutation unit save the new chromosome if its fitness was better or equal to the existing chromosome. The retained chromosome would then be mutated and used to reconfigure the virtual FPGA. The mutation rate was set by the NIOS processor with the two bit mutation rate lines. At any stage the chromosome could be read by using the serial clock and serial data lines after resetting the chromosome counter.

Mutation rate

The mutation rate was selected by two bits controlled by the NIOS processor which caused the mutation unit to invert from one to four bits in the chromosome. The mutation rate was inversely proportional to the fitness, thus as the fitness level increased, the mutation rate decreased. The random number generator produced four random 9-bit numbers ranging from 0 and 511; these were used to select which bit in the 383-bit chromosome would be mutated. It should be noted that the length of the chromosome was smaller than the random number, thus it was possible that a mutation might not occur. Thus the mutation produced a mutation rate, as well as a mutation probability as described in chapter eight. These values as shown in Table 9-2 were produced using Equation 8-1 and Equation 8-2.

mutation bits	mutation probability	maximim mutation rate
1	75%	0.26%
2	94%	0.52%
3	98%	0.78%
4	99.6%	1.04%

Table 9-2. Mutation rates for reducing layer virtual FPGA.

9.5 Results

Initially two experiments were performed, the first using a software simulation of the ball and beam, the second using a hardware simulation of the ball and beam. The fitness was determined by the length of time the ball was kept balanced before hitting an end-stop. At the beginning of each test the ball and beam were placed at their reset positions, with the ball starting at rest at the left of the beam, and the beam tilted at an angle of 20 degrees to the left above the horizontal plane.

After the initial tests the execution speed of the hardware simulation was improved by increasing the hardware simulation clock from 5MHz to 50MHz, and reducing the delay times within the virtual FPGA. A third set of tests were performed to evaluate this new system.

The results compare the hardware and software simulation performances. The results are described in the following order:

- validation of the hardware and software simulation;
- the evolved behaviour of the ball and beam as the evolution progressed;
- a comparison of the relationship between fitness and the generation number;
- a comparison of the time taken to evolve successful solutions;
- the performance of the 50MHz hardware simulation;
- a comparison of the time taken to evolve a successful solution for the 5 MHz and 50MHz hardware simulations and the software simulation.

In the following tables within this chapter, the numbers relate to the beam states. These states are comprised of the ball speed with three values: 0 the ball is moving to the left, 1 the ball is stopped or slow moving and 2 the ball is moving to the right. The ball position had nineteen values: 0 indicating the ball is at the left most position and 18 the ball at the right most position. The beam position had ten values: 0 indicating the beam was at its maximum right angle, and 9 showing the beam at its maximum left angle. The

virtual FPGA had a single binary output which was used to drive the beam. This gave the beam motor two speeds, maximum forward and maximum reverse, with 1 indicating the motor was driven to the left, and 0 the motor was driven to the right.

9.5.1 Validation of the Hardware and Software Simulation

In order to confirm that the hardware and software simulation acted in a similar manner, a simple test was performed. The simulation was run on both systems starting from the normal reset position. The beam was kept stationary whilst the ball ran down the beam until it reached the end-stop. A recording of the beam states (ball position, ball speed, beam position), was taken whenever a discrete change in one of these states within the beam simulation occurred. Concurrently the time at each discrete change in the beam state was also recorded. The comparison of the software and hardware simulation shown in Table 9-3 indicates that the simulations were identical in nature.

software simulation				hardware simulation 5MHz				time
ball positn	ball speed	beam positn	time (ms)	ball positn	ball speed	beam positn	time (ms)	between sensors
0	1	9	0	0	1	9	0	
0	2	9	184	0	2	9	184	
1	2	9	229	1	2	9	229	229
2	2	9	445	2	2	9	445	216
3	2	9	581	3	2	9	581	136
4	2	9	656	4	2	9	656	75
5	2	9	715	5	2	9	715	59
6	2	9	775	6	2	9	775	60
7	2	9	815	7	2	9	815	40
8	2	9	850	8	2	9	850	35
9	2	9	881	9	2	9	881	31
10	2	9	923	10	2	9	923	42
11	2	9	947	11	2	9	947	24
12	2	9	970	12	2	9	970	23
13	2	9	991	13	2	9	991	21
14	2	9	1015	14	2	9	1015	24
15	2	9	1033	15	2	9	1033	18
16	2	9	1050	16	2	9	1050	17
17	2	9	1070	17	2	9	1070	20
18	2	9	1085	18	2	9	1085	15

Table 9-3. Comparison of the characteristic of the simulation.

From the recorded data it can be seen that the ball slowly moved to the right increasing in speed as the ball progressed down the beam. This can be seen in the column time between sensors, showing the reducing amount of time that it took for the ball to pass the sensors, starting off slowly and then increasing in speed due to the increased slope of the beam and the pull of gravity. Note there was a seemingly incorrect variation in time between some of the sensors. This was because the sensors were not evenly spaced, for instance the sensors at position one, eight and seventeen had a different spacing than the others.

9.5.2 Behaviour of the ball and beam

The evaluation of each individual began with the beam and ball in the reset position. The motion and resulting behaviour of the ball and beam were observed and a recording of the ball speed, ball position, beam position and time was stored by enabling the beam-ball graphics on the graphical user interface. From this observed and recorded data it was determined that there were five stages of evolution, each linked to a level of fitness. The stages were:

- a fitness level at one second;
- a fitness level at two seconds;
- a fitness level at ten seconds;
- a fitness level between twenty and fifty seconds;
- a successful evolution.

When analysing the beam motion it should be remembered that the beam could not be stopped; it had to move either right or left. The following tables show a summary of the ball and beam motion, as not all the data can be shown due to the length of data recorded.

First evolution stage

The first stage of evolution had individuals that either drove the beam continuously to the left, causing the ball to roll to the left end-stop within 300ms, or the beam was held at its maximum right angle with the ball rolling to the right end-stop within 1 second. The recording for this stage are shown in Table 9-4.

ball hitting left end-stop					ball hitting right end-stop				
ball positn	ball speed	beam positn	motor drectn	time (ms)	ball positn	ball speed	beam positn	motor drectn	time (ms)
0	1	9	1	1	0	1	9	0	2
0	1	8	1	16	0	2	9	0	184
0	0	8	1	157	1	2	9	0	229
0	0	7	1	176	2	2	9	0	445
0	0	7	1	264	3	2	9	0	581
					4	2	9	0	656
					5	2	9	0	715
					6	2	9	0	775
					6	2	9	0	776
					7	2	9	0	815
					8	2	9	0	850
					9	2	9	0	881
					10	2	9	1	923
					11	2	9	0	947
					12	2	9	0	970
					13	2	9	0	991
					14	2	9	0	1015
					15	2	9	0	1033
					16	2	9	0	1050
					17	2	9	0	1070
					17	2	9	0	1071
					18	2	9	0	1085
					18	2	9	0	1096

Table 9-4. Stage I of the evolutionary process showing the ball and beam motion.

Second evolution stage

The second stage of evolution showed a jittering of the beam at static beam positions as can be seen in Table 9-5. For example, the beam would jitter left then right around the nine and eight beam position, or around the beam eight and seven positions. As mentioned previously the beam has only two speeds, left and right, and can not be stopped thus the evolution is finding a means to slow down the speed and travel of the ball by jittering the beam. The virtual FPGA would produce a constant direction in the motor until an input was changed, such as the beam position. The first evolved circuits thus triggered off two beam positions to alternate the motor direction. The jitter in the beam would slow the ball down but it would still quickly reach an end-stop within two seconds.

ball hitting left end-stop					ball hitting right end-stop				
ball positn	ball speed	beam positn	motor drectn	time (ms)	ball positn	ball speed	beam positn	motor drectn	time (ms)
0	1	9	0	1	0	1	9	0	1
1	2	8	1	264	1	2	8	1	566
1	2	9	1	497	2	1	7	0	738
2	2	8	1	566	2	2	7	1	1042
2	2	7	1	681	3	2	8	0	1307
3	2	8	1	1631	4	2	9	1	1508
4	2	6	0	1765	5	2	8	1	1596
4	2	7	1	1914	6	2	8	0	1676
4	1	5	0	2077	7	2	8	1	1727
4	0	6	1	2098	8	2	8	1	1772
4	0	5	0	2137	9	2	8	0	1810
4	0	5	0	2181	10	2	8	0	1861
4	0	6	1	2182	11	2	8	1	1891
3	0	6	0	2192	12	2	8	1	1917
2	0	6	1	2193	13	2	8	1	1942
1	0	6	1	2307	14	2	8	1	1970
1	0	5	0	2433	15	2	8	1	1991
0	0	6	1	2458	16	2	7	1	2012
0	0	5	0	2479	17	2	7	1	2033
0	0	6	1	2492	18	2	7	0	2050

Table 9-5. Stage II of the evolutionary process showing the ball and beam motion.

Third evolution stage

The next significant improvement was the third evolution stage, where the ball would be slowed down by the beam jittering around several different beam positions depending on the current position of the ball. The beam would begin to follow the balls motion, thus increasing the amount of time that the ball would spend slowly moving or stopped. This can be seen in Table 9-6 where the beam is jittering around positions nine to eight, then eight to seven, then seven to six. The virtual FPGA was now beginning to include the ball position data as well as the beam position data as part of its output determination. However eventually control of the ball would be lost and the movement of the beam would not be enough to prevent the ball from hitting an end-stop. The fitness level for this stage was between two and ten seconds.

ball hitting left end-stop					ball hitting right end-stop				
ball positn	ball speed	beam positn	motor drectn	time (ms)	ball positn	ball speed	beam positn	motor drectn	time (ms)
0	2	9	1	184	0	2	9	1	184
1	1	9	0	497	1	1	8	0	380
2	1	7	0	738	2	2	8	1	681
2	2	7	1	1042	2	1	8	0	1181
3	2	8	1	1307	3	2	7	1	1370
3	1	7	0	1500	3	1	8	0	1631
4	2	6	1	1874	4	2	6	1	1874
5	2	7	1	2218	4	2	7	1	2088
5	1	7	0	2437	5	1	6	0	2335
6	2	6	1	2764	5	2	7	1	2521
7	2	5	1	3217	6	1	6	0	2787
7	2	5	1	3552	6	2	5	1	3152
8	2	5	1	3966	7	1	6	0	3319
9	2	5	1	4422	7	2	5	1	3552
9	2	3	1	4720	8	1	5	0	3710
10	1	5	0	4887	8	2	5	1	3966
10	2	3	1	5168	9	1	4	0	4205
11	2	4	1	5490	9	2	5	1	4422
12	2	2	0	5792	9	2	3	1	4720
12	2	4	1	6009	10	2	5	1	4947
13	2	2	0	6278	10	2	3	1	5168
13	1	2	1	6555	11	2	4	1	5490
11	0	1	0	6832	12	2	3	0	5688
6	0	2	1	7109	13	2	4	0	5899
1	0	2	1	7235	17	2	4	1	6171
0	0	2	1	7253	18	2	4	1	6205

Table 9-6. Stage III of the evolutionary process showing the ball and beam motion.

Fourth evolution stage

The fourth stage ranging from ten to fifty seconds had the beam moving in such a manner as to quickly bring the ball to a slow speed, with the beam position following the ball position as shown in Table 9-7. The beam aligned itself with the ball such that the ball was roughly balanced, remaining relatively stationary for longer periods of time. Eventually the ball would move from this static point to another position on the beam. When this occurred the beam would rapidly track the ball's motion, slowing it down and bringing the ball once again to a relatively stationary state. This pattern would repeat for long periods of time until the ball would move closer to an end-stop. In this position any movement towards the end-stop could not be immediately countered by the beam so the ball would touch the end-stop and the fitness evaluation would end.

ball positn	ball speed	beam positn	motor drectn	time (ms)	ball positn	ball speed	beam positn	motor drectn	time (ms)
0	1	9	0	2	0	1	9	0	2
4	2	6	1	1874	2	2	8	1	896
8	1	5	0	3710	3	1	8	0	1631
8	1	6	1	3869	5	1	7	0	2437
11	1	2	0	5670	7	2	5	1	3217
10	2	3	1	7570	8	2	4	0	4094
10	1	3	0	9738	9	2	5	1	4813
14	2	2	0	11394	11	1	2	0	5670
17	0	1	0	13248	8	1	5	0	6420
18	2	0	1	15015	9	1	5	0	7287
17	1	1	0	16872	11	1	2	0	8078
17	2	0	1	18800	10	2	3	1	9674
17	2	0	1	20671	12	2	4	1	10554
16	1	2	1	24299	14	2	3	1	11305
17	1	1	0	26324	17	1	1	0	12040
16	0	2	0	28223	17	2	0	1	13788
17	0	1	0	30075	17	1	1	0	14629
17	1	2	1	31816	18	1	0	1	15150
17	0	1	0	33764	16	1	1	0	16147
17	0	1	0	36285	17	2	1	1	16958
17	1	2	1	37500	18	0	0	1	17597
17	2	1	1	39393	17	2	1	1	18527
18	0	0	1	41151	18	0	0	1	19504
17	1	2	1	43296	16	1	1	0	20021
17	1	0	0	45117	16	1	2	1	20100
18	1	0	1	45887	18	2	2	0	21726

Table 9-7. Stage IV of the evolutionary process showing the ball and beam motion.

Fifth evolution stage

The fifth stage showed a sudden step change to the maximum fitness. The traits for this stage had the ball slowly moving around a section of the jittering beam away from either end-stop. The ball would stay in this position for a long period of time before it gained enough momentum to move towards the opposite side of the beam. The beam would move to counteract this motion and bring it back to its original position. This pattern would repeat itself without the ball reaching an end-stop until the maximum fitness value was reached giving a successful evolution.

9.5.3 Comparison between the Software and Hardware Simulation

Comparison of improvement in fitness level with the number of generations

The graphs of the fitness level relative to the number of generations for both the software simulation and hardware simulation are shown in Figure 9-15 and Figure 9-16 respectively. These graphs show that the evolution process is similar in both methods, with an average number of generations to a successful evolution of approximately

100,000 generations. These graphs show the step characteristics of the evolutionary process as the individuals evolve through the various evolutionary stages.

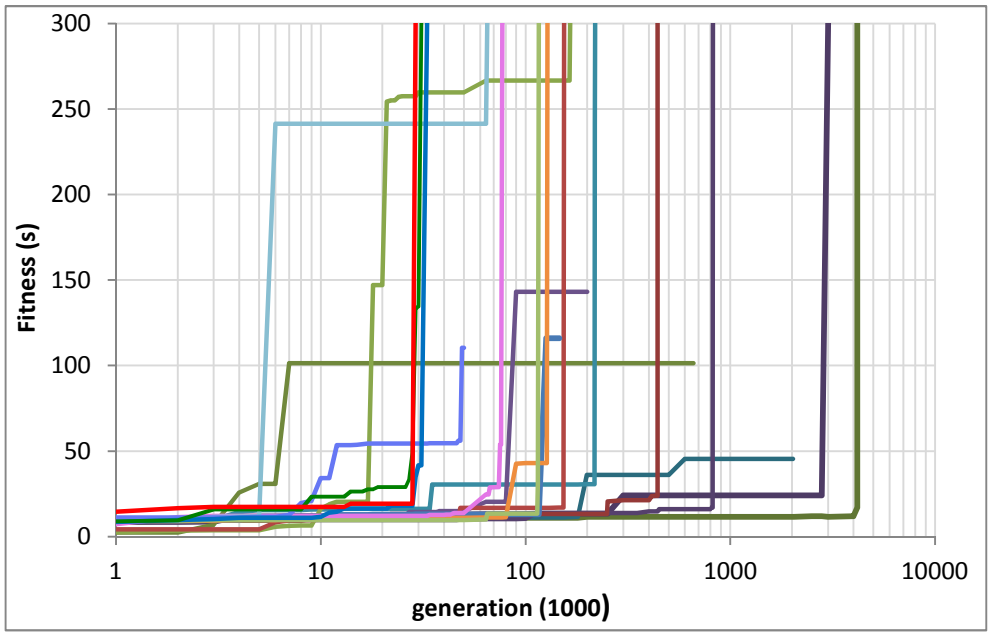


Figure 9-15. Fitness relative to generation for the software simulation.

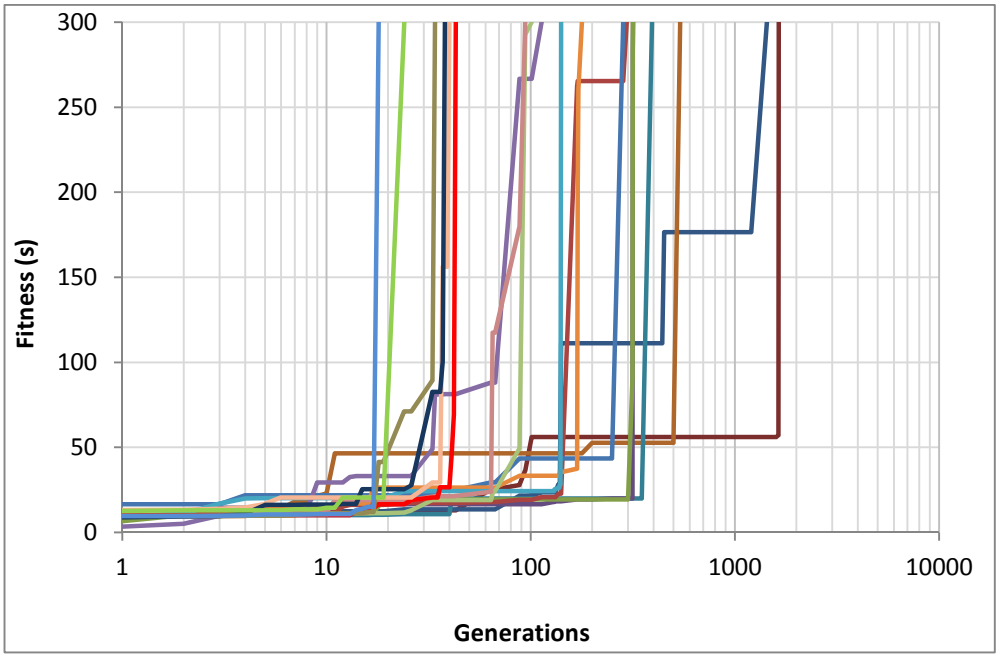


Figure 9-16. Fitness relative to generation for the hardware simulator operating at 5MHz.

Comparison of improvement in fitness level with evolutionary time

The graphs relating the fitness level to the evolutionary process time for both the software and hardware simulation are shown in Figure 9-17 and Figure 9-18

respectively. It can be seen that the average time for a successful evolution using the software simulation is approximately 50,000 seconds or 14 hours, whereas the hardware simulation average time for a successful evolution was approximately 750 seconds or 13 minutes. The hardware simulation had a speed improvement over the software simulation of approximately 70 times.

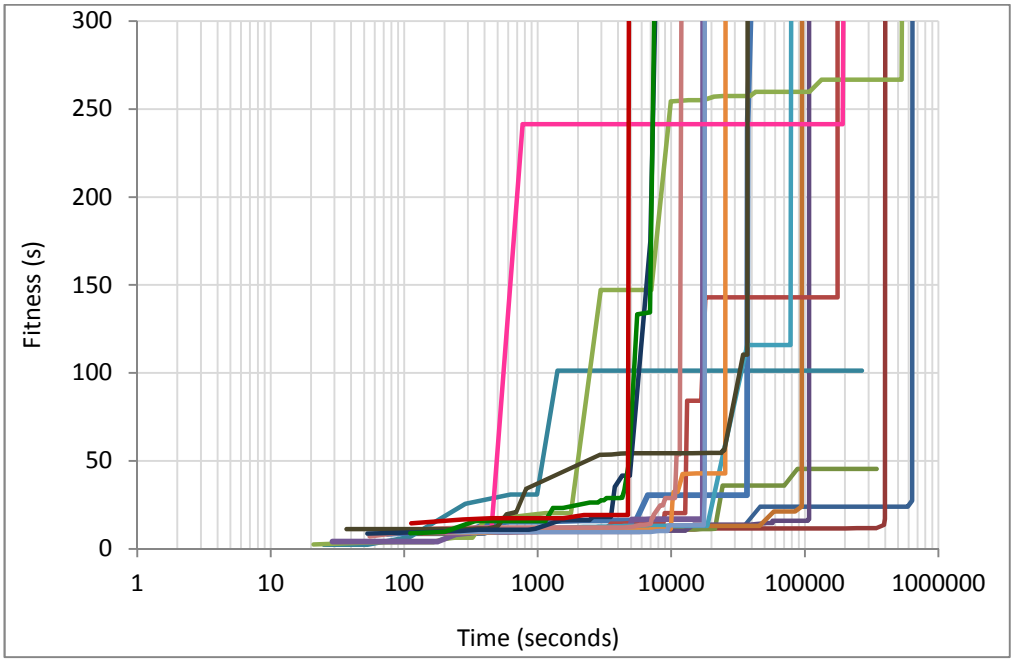


Figure 9-17. Fitness relative to evolutionary time for the software simulation.

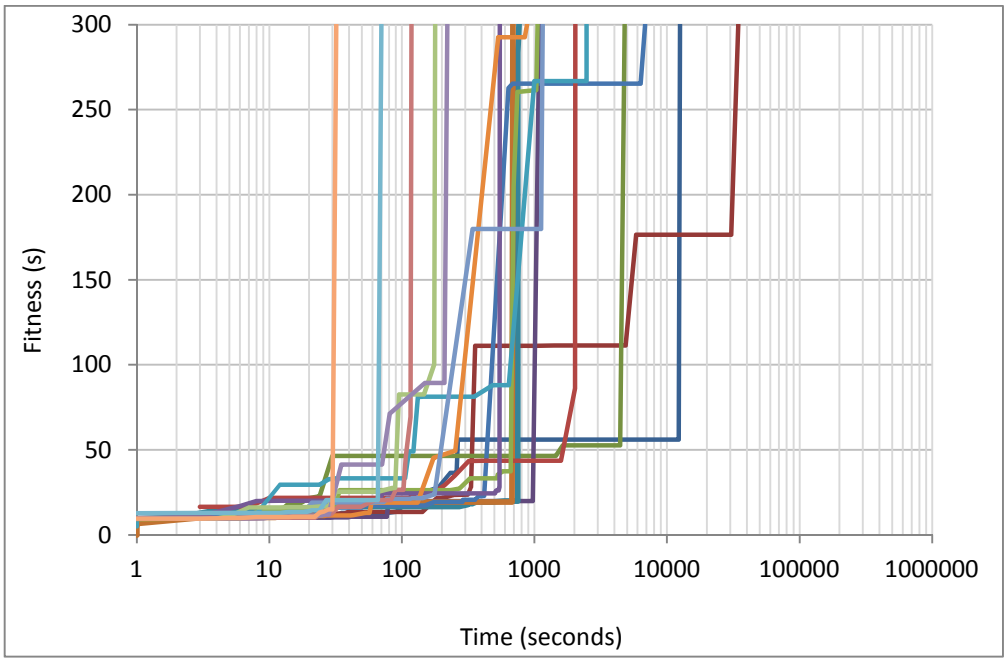


Figure 9-18. Fitness relative to evolutionary time for the hardware simulation with 5MHz clock.

9.5.4 Comparison with Hardware Simulation Running at 50MHz

The maximum speed of the hardware simulation was limited to 5MHz, due to the 100ns delay time within the virtual FPGA rather than the execution of the hardware simulation itself. In previous experiments using a software simulation, a delay of 100ns between the inputs and outputs of the virtual FPGA was not important as the delays in the software simulation were far in excess of that figure. However the speed improvement of the hardware simulation now meant that the virtual FPGA was the limiting factor. In order to increase the speed of the hardware simulation, the internal clocking of the virtual FPGA was removed as shown in Figure 9-19, and the clock speed of the hardware simulation increased to 50MHz. In theory this should give a speed improvement of one order of magnitude over the 5MHz hardware. This improvement could be slightly reduced as the time taken to perform the selection and reproduction tasks of the genetic algorithm were not changed.

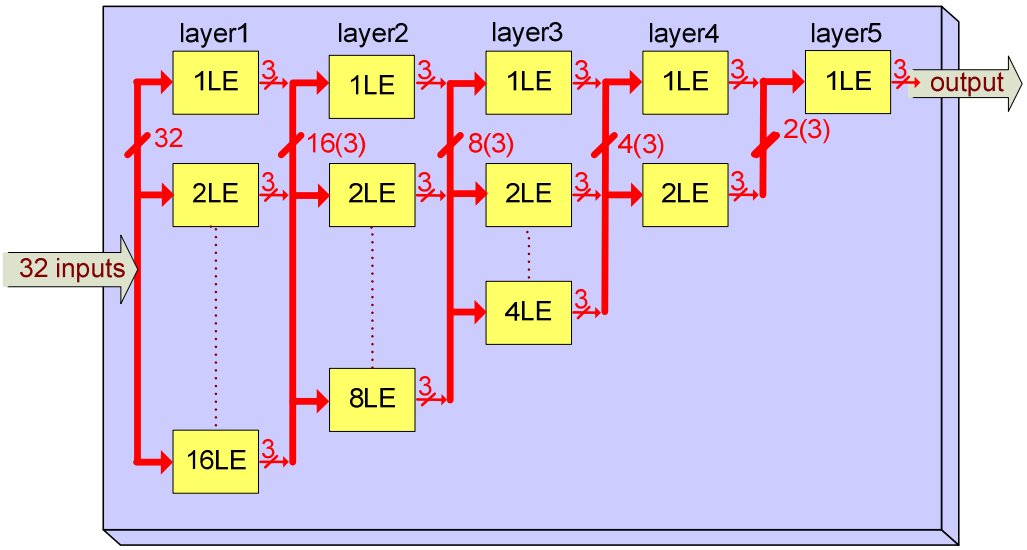


Figure 9-19. Architecture for the reducing layer virtual FPGA with no internal clock.

The graph of the fitness relative to the number of generations is shown in Figure 9-20. It can be seen that the simulation was working in a similar manner as the software and 5MHz simulations with the step change in fitness, and the average number of generations required to reach a successful evolution at approximately 100,000 generations.

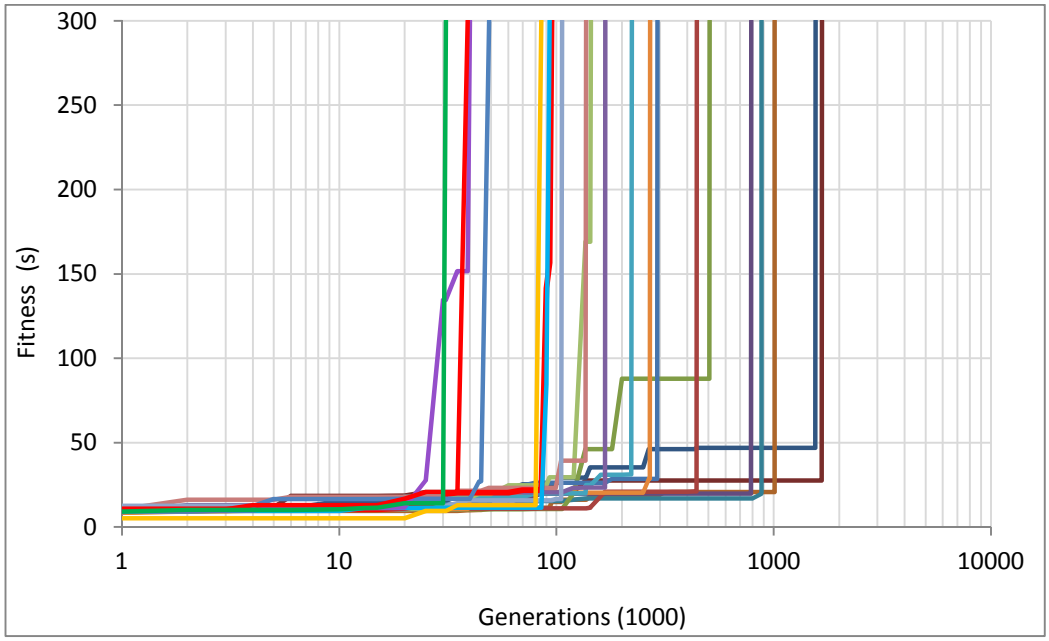


Figure 9-20. Fitness relative to generation for the hardware simulation operating at 50MHz.

The graph relating fitness level to time is shown in Figure 9-21. The time taken for a successful individual to evolve was 110 seconds; this was 700 times faster than the software simulation.

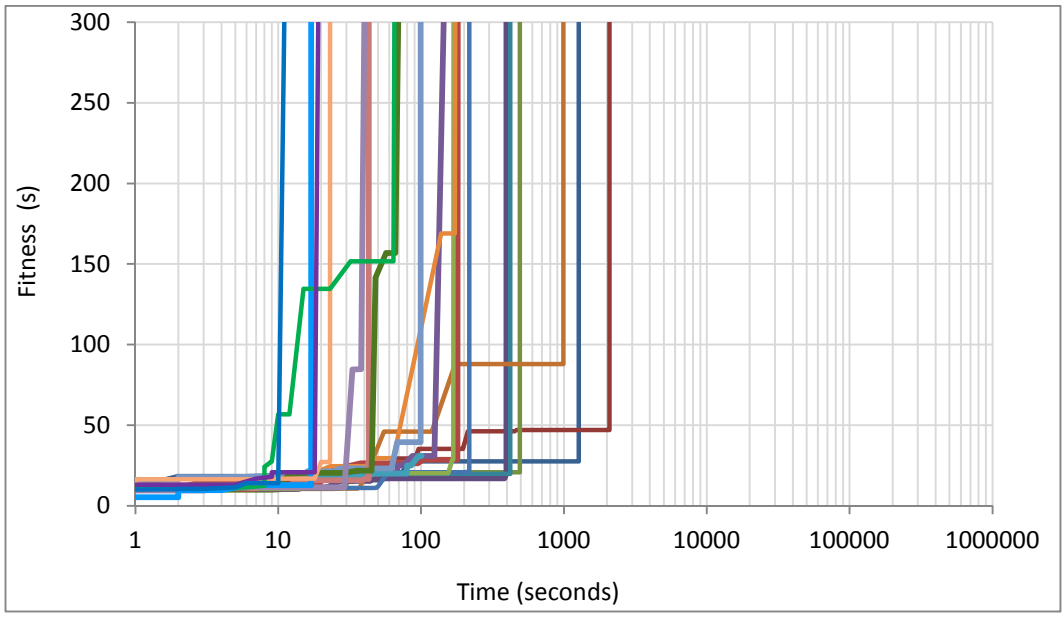


Figure 9-21. Fitness relative to evolutionary time for the hardware simulation operating at 50MHz.

A comparison of the time required to reach a successful solution at approximately 35,000 generations for each of the three types of simulation is shown in Figure 9-22. It can be seen that the hardware simulation running at 50MHz was the fastest at 11 seconds, the hardware simulation running at 5 MHz was 110 seconds and the software simulation was 8,000 seconds

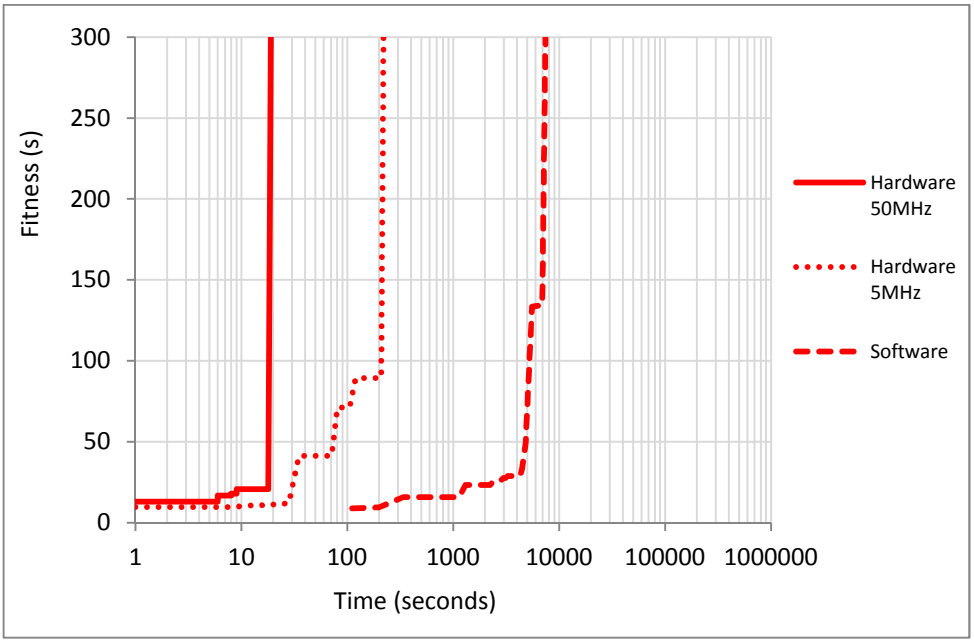


Figure 9-22. Comparison of time taken to reach a successful evolution at 35,000 generations for the three simulations.

9.6 Conclusions

It can be seen that a simulation can be designed to run in hardware using standard Verilog coding commands. The simulation needs to be modified so that floating point arithmetic and trigonometric functions are removed. The timing of the simulation must be carefully designed to interface with the genetic algorithm, and the fitness evaluation.

A hardware simulation replicating a balancing beam has been successfully implemented. This simulation has been used in a hardware genetic algorithm to evolve a virtual FPGA that was capable of balancing the ball on the beam for more than five minutes. A comparison between identical software and hardware simulations was performed with both systems behaving in an identical manner. However it was found that the hardware simulation could evolve successful circuits over 700 times faster than the software simulation.

A conference paper was accepted for this section with recommendation for best paper award, and journal publication. The Verilog code, C code and graphical user interface can be found in the CD accompanying this thesis.

Chapter 10

Chapter 10: Conclusions and Future Research

10.1 Summary

This thesis has added to the body of knowledge in the field of evolutionary robotics by developing the novel concept of using evolutionary capable lookup tables as robotic controllers. The lookup table links the current state of the robot to a desired robotic action for that state. The lookup table is suitable for evolution as shown in two examples using firstly a mobile inverted pendulum, and secondly a ball-balancing beam. Advancements in knowledge have also been performed in the subfield of evolvable hardware using both fixed layer and reducing layer architectures for an evolved virtual FPGA used as a robotic controller for a ball-balancing beam. In this case the current state of the robot was connected to the input of the virtual FPGA, and the evolved circuits produced an output to control the robot. Finally the concept of moving the simulation used for fitness evaluation of individuals from software to hardware has been performed with a corresponding decrease in evolution completion time of approximately 700 times. The following sections summarise these points.

10.1.1 Thesis Précis

Chapter one introduced the subject area of evolutionary robotic controllers. A list of research questions this thesis answers was stated, with a brief explanation of how they would be answered. Chapter two described the techniques employed in genetic algorithms including reproduction, selection, fitness evaluation, and the application of lookup tables. Chapter three explained the requirements for evolving hardware using techniques such as genetic compilers, genetic programming and Cartesian based virtual FPGAs to overcome the problems associated with evolvable hardware. Chapter four reviewed the area of major research in evolutionary robotics using artificial neural networks and fuzzy logic controllers. Basic concepts and their adaption for evolutionary robotics were also explained. Chapter five summarised recent research in the two systems that were evaluated in this thesis, including the mobile inverted pendulum and

the ball-balancing beam. Chapter six provided the derivation of the mathematical models and simulations for the mobile inverted pendulum and ball-balancing beam. Chapter seven detailed how lookup tables could be evolved for robotic controllers. This chapter provided two examples of how they could be implemented; firstly by using a two-dimensional array to control a mobile inverted pendulum, and secondly by using a three-dimensional array to control a ball-balancing beam. Chapter eight presented the fixed layer virtual FPGA that was evolved using a hardware genetic algorithm to control the ball-balancing beam. Chapter nine illustrated how the robotic simulation could be moved from hardware to software, with a large improvement in the evolutionary completion time. This chapter also presented the reducing layer virtual FPGA with a reduced chromosome length and more powerful functionality than the fixed layer virtual FPGA.

10.1.2 Lookup Tables

Can a lookup table be evolved to function as a robotic controller?

Two systems were developed to evaluate the use of evolved lookup tables for robotic controllers. The first evolved a two dimensional lookup table that was used to control a mobile inverted pendulum. The axes of the lookup table were related to the pendulum's angle and angular velocity, while the parameters within the lookup table provided the motor torque and direction that would maintain the pendulum in balance. The software genetic algorithm used tournament selection, full crossover and creeping mutation. A simulation of the mobile inverted pendulum was produced for the evaluation of fitness. How the individual was evaluated was important as it determined the final behaviour of the pendulum. The fitness level was dependant on how long the individual could maintain the pendulum in balance while keeping within ±0.5 meters of its starting position. Multiple starting angles were used when testing each individual. The pendulum evolved a behaviour that would swiftly bring the pendulum upright from an initial lean, and then oscillate around a fixed angular and horizontal position. It was found that a successful evolved lookup table could keep the pendulum balanced for more than 250 seconds.

The second system was a three-dimensional lookup table used as a controller for a ball-balancing beam which was evolved using a similar software genetic algorithm as that used for the mobile inverted pendulum described above (except for mutation rates). The

three axes of the lookup table were related to the beam position, ball position and ball speed. The parameters inside the lookup table gave the required motor speed and direction such that the beam would move in a motion that would keep the ball in balance. A simulation of the beam was used to evaluate the fitness of each individual in the population. The fitness level was dependent on how long the ball would remain in balance before hitting an end-stop. Multiple starting points were used when evaluating each individual. Several experiments were performed using a different number of motor speeds ranging from two (forward and backward), to eleven (five forward, five backward and one stopped). The motion of the ball and beam for a successful individual was recorded showing the ball had been captured in one place by the beam oscillating around two fixed positions. It was found that the evolved lookup table controller could keep the ball balanced for more than five minutes. All the experiments with a range of motor speeds and different maximum motor speeds evolved successful controllers; however those with a higher maximum motor speed and limited range of set speeds had a faster evolution time.

10.1.3 Virtual FPGA

Can a virtual FPGA be evolved to function as a robotic controller?

An evolutionary capable fixed layer virtual FPGA was constructed to act as a controller for the ball-balancing beam. The virtual FPGA had thirty-two inputs to read the ball beam states of ball position (nineteen), ball speed (three), and beam position (ten). It had one output which was used to select a forward or reverse motor speed to drive the beam. The virtual FPGA was based on a Cartesian architecture with a four column by sixteen row, two-dimensional array of logic elements The genetic algorithm was constructed in hardware using mutation only, and was used to evolve the virtual FPGA by modifying its configuration bit stream. This genetic algorithm was chosen so that it could be used in a FPGA with limited resources, and although not as powerful as a full genetic algorithm incorporating crossover, it was still able to evolve controllers. A graphical user interface was constructed to give a visual representation of the motion of the ball and beam, and to provide control and data logging. It was found that an evolved virtual FPGA circuit could control the motion of the beam to maintain the ball in balance for more than five minutes. The fitness improved in large steps with a final jump in fitness to a successful solution. An analysis of the ball and beam motion found

that once the ball had been slowed down from its initial starting roll, it could be brought to balance by the beam oscillating between two points placing the ball in a stable state.

10.1.4 Hardware Simulation

Can the genetic algorithm's simulation be implemented in hardware and benefit the evolutionary process?

The time required for the evolutionary process to find a successful individual is largely dependent on the time taken for the fitness evaluation of an individual. This is normally performed in software; however if it could be implemented in hardware then a large improvement in the evolutionary process would occur. The mathematical model of the ball-balancing beam was converted to integer values so that it could be created as a hardware circuit implemented in a resource limited FPGA. Two simulations using this model were created; the first was a software simulation and the second a hardware simulation. The hardware simulation included a simulation unit which contained the simulation equations in hardware as well as a fitness calculation unit that could determine how long the ball had remained balanced. The controller used for the experiment was a reducing layer virtual FPGA which had a reduced configuration bit stream, and more powerful function operators than the fixed layer virtual FPGA used in previous experiments. The reducing layer virtual FPGA was based on a Cartesian architecture two-dimensional array of logic elements. There were five columns of logic elements with each column having a reducing number of rows reducing from sixteen elements in the first column to one in the fifth column. The interface between the simulation and virtual FPGA was carefully designed to ensure the timing between these two systems was correct. Tests carried out on the software and hardware simulations showed that they performed identically; however the hardware simulation had a 700 times speed improvement over the software simulation. The evolutionary process of the ball and beam motion was carefully studied, showing five stages of evolutionary learning, as the beam evolved to balance the ball. The result of these experiments showed that a genetic algorithm's simulation could be executed in hardware with a significant improvement in evolutionary completion time.

10.2 Future Research

The thesis has investigated three significant questions in the field of evolutionary robotics. The answers to these questions have been discussed, however these answers

lead to more questions that can be investigated in the future. These questions are explained next.

How does a lookup table compare with virtual FPGA?

The performance of a lookup table and a virtual FPGA for an evolved robotic controller will be compared using a simulation of the ball-balancing beam, and the hardware genetic algorithm for the genetic process. The lookup table will be generated in RAM with the RAM address linked to the ball position, ball speed and beam position while the data at each address will relate to the desired motor speed. A chromosome will be the contents of the RAM table with the hardware genetic algorithm modifying the contents of the data within the RAM. The data size of the RAM can be modified from one to three bits to give a range of speeds.

How does a chromosome evolved on a simulation of the beam perform on a real beam?

The physical beam developed by students at AUT University was never fully completed as there were problems with the ball position sensors, and the motor drivers. Also the beam controller was an Atmel Mega128 microcontroller, rather than a FPGA device. The construction of a new curved ball-balancing beam is now in the planning stage by technicians at the University using: the Terasic FPGA DE0-Nano board as the controller; a high torque DC motor with over 360^0 per second angular velocity; and a magnetic and resistive ball position sensor. When this is completed, the simulations will be modified for the new system and the evolution run again. The evolved virtual FPGA will be able to be directly downloaded to the DE0-Nano board FPGA for an evaluation between the simulation and physical beam.

How does the fixed layer virtual FPGA compare with the reducing layer virtual FPGA?

The fixed layer virtual FPGA and the reducing layer FPGA have both been used successfully in different experiments; however their performances have not been directly compared. An evaluation of each system will be made using the simulation for the balancing beam and the hardware genetic algorithm. The virtual FPGAs will be evaluated on how quickly they can be evolved, and how well they control the motion of the ball and beam.

How does a software genetic algorithm compare with a hardware genetic algorithm?

A hardware genetic algorithm has been used to evolve the virtual FPGAs used for robotic controllers. A software genetic algorithm running on the NIOS processor will be created and compared with the hardware genetic algorithm using the simulation for the ball-balancing beam and a virtual FPGA. The two genetic algorithms will be evaluated on how quickly they can produce a successful individual, and how well they control the ball and beam.

Can a software model of the virtual FPGA be created and how will it compare with a lookup table?

The virtual FPGA has been constructed in hardware and has been shown to be able to be evolved to create a robotic controller. A software model of the hardware virtual FPGA will be created and its ability to be evolved for a robotic controller will be evaluated against that of a lookup table.

References

[1] G. Dudek. (2011). *What is a robot*. Available: http://aaai.org/AITopics/Robots

[2] M. Mitchell and C. E. Taylor, "Evolutionary Compuation: An Overview," *Annual Review of Ecological Systems,* vol. 30, pp. 595-616, 1999.

[3] J. Holland, *Adaption in Natural and Artifical Systems*: MIT Press, 1975.

[4] J. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*: The MIT Press, , 1992.

[5] J. R. Koza, *Genetic Programming: on the programming of computers by means of natural selection*: The MIT Press, 1992.

[6] I. Rechenberg, "Cybernetic solution path of an experimental problem," Royal Aircraft Establishment (UK), Ministry of Aviation, Farnborough1965.

[7] I. Rechenberg, *Evolutionsstrategie; Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*: Frommann-Holzboog, 1973.

[8] H.-P. Schwefel, *Numerische Optimierung von Computor-Modellen mittels der Evolutionsstrategie: Mit einer vergleichenden Einführung in die Hill-Climbing- und Zufallsstrategie*: Birkhäuser; 1. Aufl edition, 1977.

[9] L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artifical Intelligence Through Simulated Evolution*. New York: Wiley Publishing, 1966.

[10] L. J. Fogel, *Intelligence Through Simulated Evolution: Forty Years of Evolutionary Programming*: Wiley-Interscience 1999.

[11] A. E. Eiben and C. H. M. van Kemende, "Diagonal crossover in genetic algorithms for numerical optimization," *Control and Cybernetics,* vol. 26, pp. 447-65, 1997.

[12] P. Rahila and M. M. Raghuwanshi, "Multi-Objective Optimization Using Multi Parent Crossover Operators," *Emerging Trends in Computing and Information Sciences,* vol. 2, pp. 99-105 2011.

[13] H. Mühlenbein, "How genetic algorithms really work: I.Mutation and hill-climbing," in *Proceedings of the Second Conference on Parallel Problem Solving from nature*, Brussels 1992, pp. 15-26.

[14] T. Back, "Selective pressure in evolutionary algorithms: a characterization of selection mechanisms," in *Evolutionary Computation, 1994. IEEE World*

Congress on Computational Intelligence., Proceedings of the First IEEE Conference on, 1994, pp. 57-62 vol.1.

[15] B. A. Julstrom, "It's all the same to me: revisiting rank-based probabilities and tournaments," in *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, 1999, p. 1505 Vol. 2.

[16] I. Harvey, "Artificial Evolution: A Continuing SAGA," in *Evolutionary robotics : from intelligent robotics to artificial life ER 2001 : evolutionary robotics. International symposium,* , Tokyo , JAPAN,, 2001, pp. 94-109.

[17] M. Okura, H. Matsumoto, A. Ikeda, and K. Murase, "Artifical evolution of FPGA that controls a Miniature Mobile Robot Khepera," in *SICE Annual Conference in Fukui*, Fukui University, Japan, 2003.

[18] P. J. Hancock, "An Empirical Comparison of Selection Methods in Evolutionary Algorithms," *Selected Papers From AISB Workshop on Evolutionary Computing, SpringerVerlag,* pp. 865: 80-94., 1994.

[19] S. Gupta, "Relative Fitness Scaling for Improving Efficiency of Proportionate Selection in Genetic Algorithms," presented at the GECCO Genetic and Evolutionary Computation Conference, Montreal Canada, 2009.

[20] S. Legg, M. Hutter, and A. Kumar, "Tournament versus fitness uniform selection," in *Evolutionary Computation, CEC2004. Congress on*, 2004, pp. 2144-2151 Vol.2.

[21] M. Hutter, "Fitness uniform selection to preserve genetic diversity," in *Evolutionary Computation, CEC '02. Proceedings of the Congress on*, 2002, pp. 783-788.

[22] D. Whitley, S. Rana, and R. Heckendorn, "Island model genetic algorithms and linearly separable problems," in *Evolutionary Computing*, ed, 1997, pp. 109-125.

[23] H. H. Lund and O. Miglino, "From simulated to real robots," in *Evolutionary Computation, Proceedings of IEEE International Conference on*, 1996, pp. 362-365.

[24] N. Jakobi, P. Husbands, and I. Harvey, "Noise and the reality gap: The use of simulation in evolutionary robotics," presented at the Third European Conference on Artificial Life, ECAL '95, Granada, Spain in 1995.

[25] M. Orazio, H. H. Lund, and S. Nolfi, "Evolving Mobile Robots in Simulated and Real Environments," *Artificial Life and Robotics,* vol. 2, pp. 417--434, 1996.

[26] M. S. Wilson, C. M. King, and J. E. Hunt, "Evolving hierarchical robot behaviours " *Robotics and Autonomous Systems,* vol. 22, pp. 215-230, 1997.

[27] T. Dean and M. Boddy, "An analysis of time-dependent planning," presented at the In Proceedings of Association for the Advancement of Artificial Intelligence, St. Paul, Minnesota, 1988.

[28] J. J. Grefenstette and C. L. Ramsey, "An Approach to Anytime Learning," *Proceedings of the Ninth Int. Machine Learning Workshop,* pp. 189-195, 1992.

[29] J. Walker and M. Wilson, "Lifelong evolution for adaptive robots," in *Intelligent Robots and Systems, IEEE/RSJ International Conference on*, 2002, pp. 984-989 vol.1.

[30] G. B. Parker and G. E. Fedynyshyn, "Enhancing embodied evolution with punctuated anytime learning," in *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, 2007, pp. 190-195.

[31] G. Capi, S. Kaneko, K. Mitobe, L. Barolli, and Y. Nasu, "Optimal trajectory generation for a prismatic joint biped robot using genetic algorithms," *Robotics and Autonomous Systems,* vol. 38, pp. 119-128, 2002.

[32] M. Beckerleg and J. Collins, "An Analysis of the Chromosome Generated by a Genetic Algorithm Used to Create a Controller for a Mobile Inverted Pendulum," *Studies in Computational Intelligence,* vol. 76, 2007.

[33] J. Currie, M. Beckerleg, and J. Collins, "Software Evolution of a Hexapod Robot Walking Gait," in *Mechatronics and Machine Vision in Practice, M2VIP 2008. 15th International Conference on*, 2008, pp. 305-310.

[34] M. Beckerleg and J. Collins, "A GA based Controller for a Mobile Inverted Pendulum," presented at the ICARA The Third Internatoinal Conference on Autonomous Robots and Agents, Palmerston Nth, New Zealand, 2006.

[35] C. E. Thomaz, M. A. C. Pacheco, and M. M. Vellasco, "Mobile robot path planning using genetic algorithms," *Lecture Notes in Computer Science,* 1999.

[36] O. Castillo, L. Trujillo, and P. Melin, "Multiple Objective Genetic Algorithms for Path-planning Optimization in Autonomous Mobile Robots," *Soft Computing - A Fusion of Foundations, Methodologies and Applications,* vol. 11, pp. 269-279, 2007.

[37] J. Ahuactzin, P. Talbi, P. Bessière, and E. Mazer, "Using genetic algorithms for robot motion planning " *Lecture Notes in Computer Science,* vol. 708, pp. 84-93, 1993.

[38] Y. Xuesong, W. Qinghua, Y. Jia, and K. Lishan, "A Fast Evolutionary Algorithm for Robot Path Planning," in *Control and Automation, ICCA2007. IEEE International Conference on*, 2007, pp. 84-87.

[39] R. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of* vol. 2, pp. 14-23, 1986.

[40] R. A. Brooks, "Elephants Don't Play Chess," *P. Maes, ed. 'Designing Autonomous Agents: Theory and Practise from Biology to Engineering and Back',* pp. 3-15, 1990.

[41] R. A. Brooks, "A robot that walks; emergent behaviors from a carefully evolved network," in *Robotics and Automation, IEEE International Conference on*, 1989, pp. 692-4.

[42] R. A. Brooks, "Behavior-based humanoid robotics," in *Intelligent Robots and Systems, IROS 96, Proceedings of the IEEE/RSJ International Conference*, 1996, pp. 1-8 vol.1.

[43] R. Silva, H. Lopes, and C. Erig Lima, "A Compact Genetic Algorithm with Elitism and Mutation Applied to Image Recognition," in *Advanced Intelligent Computing Theories and Applications. With Aspects of Artificial Intelligence*, ed, 2008, pp. 1109-1116.

[44] R. Silva, H. Lopes, and C. Erig Lima, "A New Mutation Operator for the Elitism-Based Compact Genetic Algorithm," in *Adaptive and Natural Computing Algorithms*, ed, 2007, pp. 159-166.

[45] J. Zhang and K. Y. Szeto, "Mutation Matrix in Evolutionary Computation: An Application to Resource Allocation Problem," in *Advances in Natural Computation*, ed, 2005, pp. 112-119.

[46] K. Szeto and J. Zhang, "Adaptive Genetic Algorithm and Quasi-parallel Genetic Algorithm: Application to Knapsack Problem," in *Large-Scale Scientific Computing*, ed, 2006, pp. 189-196.

[47] K. Shiu and K. Szeto, "Self-adaptive Mutation Only Genetic Algorithm: An Application on the Optimization of Airport Capacity Utilization," in *Intelligent Data Engineering and Automated Learning* ed, 2008, pp. 428-435.

[48] P. Xia, Z. Jian, and K. Y. Szeto, "Application of Mutation Only Genetic Algorithm for the Extraction of Investment Strategy in Financial Time Series," in *Neural Networks and Brain, ICNN&B'05. International Conference*, 2005, pp. 1682-1686.

[49] H. Aguirre and K. Tanaka, "Genetic Algorithms on NK-Landscapes: Effects of Selection, Drift, Mutation, and Recombination," in *Applications of Evolutionary Computing*, ed, 2003, pp. 131-142.

[50] T. Back, "Optimal Mutation Rates in Genetic Search," presented at the Proceedings of the 5th International Conference on Genetic Algorithms, 1993.

[51] J. D. Schaffer, R. A. Caruana, L. J. Eshelman, and R. Das, "A study of control parameters affecting online performance of genetic algorithms for function optimization," presented at the Proceedings of the Third International Conference on Genetic Algorithms, George Mason University, United States, 1989.

[52] T. L. Lau and E. P. Tsang, "Applying a mutation-based genetic algorithm to processor configuration problems," in *Tools with Artificial Intelligence, Proceedings Eighth IEEE International Conference*, 1996, pp. 17-24.

[53] I. De Falco, A. Della Cioppa, and E. Tarantino, "Mutation-based genetic algorithm: performance evaluation," *Applied Soft Computing,* vol. 1, pp. 285-299, 2002.

[54] N. Raichman, R. Segev, and E. Ben-Jacob, "Evolvable hardware: genetic search in a physical realm," *Physica A: Statistical Mechanics and its Applications,* vol. 326, pp. 265-285, 2003.

[55] A. Thompson, P. Layzell, and R. S. Zebulum, "Explorations in design space: unconventional electronics design through artificial evolution," *Evolutionary Computation, IEEE Transactions on,* vol. 3, pp. 167-196, 1999.

[56] Z. Zhu, D. J. Mulvaney, and V. A. Chouliaras, "Hardware implementation of a novel genetic algorithm," *Neurocomputing,* vol. 71, pp. 95-106, 2007.

[57] L. Sekanina, T. Martinek, and Z. Gajda, "Extrinsic and Intrinsic Evolution of Multifunctional Combinational Modules," in *Evolutionary Computation, CEC2006. IEEE Congress*, 2006, pp. 2771-2778.

[58] L. Sekanina and S. Friedl, "An Evolvable Combinational Unit for FPGAS," *Computing and Informatic,* vol. 23, pp. 461-486, 2004.

[59] J. Currie, M. Beckerleg, and J. Collins, "Software evolution of a hexapod robot walking gait," *Int. J. Intell. Syst. Technol. Appl.,* pp. 382-394, 2010.

[60] H. H. Lund and J. Hallam, "Evolving sufficient robot controllers," in *Evolutionary Computation, IEEE International Conference*, 1997, pp. 495-499.

[61] H. H. Lund, "Co-evolving Control and Morphology with LEGO Robots," in *Proceedings of Workshop on Morpho-functional Machines*, 2001.

[62] A. Chavoya and Y. Duthen, "Using a genetic algorithm to evolve cellular automata for 2D/3D computational development," presented at the Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, Seattle, Washington, USA, 2006.

[63] G. Greenfield, "Evolved Look-Up Tables for Simulated DNA Controlled Robots," presented at the Proceedings of the 7th International Conference on Simulated Evolution and Learning, Melbourne, Australia, 2008.

[64] R. Y. Z., A. Krohling, Y. Zhou, and A. Tyrrell, "Evolving FPGA-based robot controllers using an evolutionary algorithm," presented at the First International Conference on Artificial Immune Systems, 2002.

[65] R. Bianco and S. Nolfi, *Evolving the Neural Controller for a Robotic Arm Able to Grasp Objects on the Basis of Tactile Sensors*, 2829 ed., 2003.

[66] D. Makaitis, "Evolving fuzzy controllers through evolutionary programming," in *Fuzzy Information Processing Society, NAFIPS . 22nd International Conference* 2003, pp. 50-54.

[67] H. Seok, K. Lee, J. Joung, and B. Zhang, "An On-Line Learning Method for Object-Locating Robots using Genetic Programming on Evolvable Hardware," *International Symposium on Artificial Life and Robotics,* pp. 321--324, 2000.

[68] D. Berenson, N. Estevez, and H. Lipson, "Hardware evolution of analog circuits for in-situ robotic fault-recovery," in *Evolvable Hardware, Proceedings NASA/DoD Conference*, 2005, pp. 12-19.

[69] C. W. Yu, J. Lamoureux, S. J. E. Wilton, P. H. W. Leong, and W. Luk, "The Coarse-Grained/Fine-Grained Logic Interface in FPGAs with Embedded Floating-Point Arithmetic Units," *International Journal of Reconfigurable Computing,* 2008.

[70] D. Lee, C. Ban, K. Sim, H. Seok, K. Lee, and B. Zhang, "Behavior evolution of autonomous mobile robot using genetic programming based on evolvable hardware," presented at the Systems, Man, and Cybernetics, IEEE International Conference, 2000.

[71] A. Thompson, "On the Automatic Design of Robust Electronics Through Artificial Evolution," in *Proceedings on the 2nd International Conference on Evolvable Systems ICES*, 1998, pp. 13-24.

[72] A. Thompson, "An evolved circuit, intrinsic in silicon, entwined with physics," in *Proceedings of the. 1st International Conference on Evolvable Systems (ICES'96)*, 1997, pp. 390-405.

[73] D. Levi and S. A. Guccione, "GeneticFPGA: evolving stable circuits on mainstream FPGA devices," in *Evolvable Hardware, Proceedings of the First NASA/DoD Workshop*, 1999, pp. 12-17.

[74] G. Hollingworth, S. Smith, and A. Tyrrell, "The Intrinsic Evolution of Virtex Devices Through Internet Reconfigurable Logic " *Lecture Notes in Computer Science,* vol. 1801, pp. 72-79, 2000.

[75] L. Dong-Wook, B. Chang-Bong, S. Kwee-Bo, S. Ho-Sik, L. Kwang-Ju, and Z. Byoung-Tak, "Behavior evolution of autonomous mobile robot using genetic programming based on evolvable hardware," in *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, 2000, pp. 3835-3840 vol.5.

[76] J. Mizoguchi, H. Hemmi, and K. Shimohara, "Production genetic algorithms for automated hardware design through an evolutionary process," in *Evolutionary Computation, IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference*, 1994, pp. 661-664 vol.2.

[77] D. Montana, R. Popp, S. Iyer, and G. Vidaver, *"EvolvaWare: Genetic Programming for Optimal Design of Hardware-Based Algorithms"*: Morgan Kaufmann, 1998.

[78] M. J. F., D. Job, and V. K. Vassilev, "Principles in the Evolutionary Design of Digital Circuits—Part I " *Genetic Programming and Evolvable Machines,* vol. 1, pp. 7-35, 2000.

[79] J. F. Miller and P. Thomson, *Cartesian Genetic Programming*, 1802 ed., 2000.

[80] G. Hollingworth, S. Smith, and A. Tyrrell, "Safe intrinsic evolution of Virtex devices," in *Evolvable Hardware, Proceedings. The Second NASA/DoD Workshop*, 2000, pp. 195-202.

[81] P. C. Haddow and G. Tufte, "An evolvable hardware FPGA for adaptive hardware," in *Evolutionary Computation, Proceedings of the 2000 Congress*, 2000, pp. 553-560 vol.1.

[82] P. C. Haddow and G. Tufte, "Bridging the genotype-phenotype mapping for digital FPGAs," in *Evolvable Hardware, Proceedings. The Third NASA/DoD Workshop*, 2001, pp. 109-115.

[83] T. Higuchi, M. Iwata, I. Kajitani, H. Yamada, B. Manderick, Y. Hirao, M. Murakawa, S. Yoshizawa, and T. Furuya, "Evolvable hardware with genetic learning," in *Circuits and Systems, ISCAS'96., 'Connecting the World', 1996 IEEE International Symposium*, 1996, pp. 29-32 vol.4.

[84] V. K. Vassilev and J. E. Miller, "Scalability problems of digital circuit evolution evolvability and efficient designs," in *Evolvable Hardware, Proceedings. The Second NASA/DoD Workshop*, 2000, pp. 55-64.

[85] T. Higuchi, M. Murakawa, M. Iwata, I. Kajitani, L. Weixin, and M. Salami, "Evolvable hardware at function level," in *Evolutionary Computation, , IEEE International Conference*, 1997, pp. 187-192.

[86] L. Sekanina, *Virtual Reconfigurable Circuits for Real-World Applications of Evolvable Hardware*, 2606 ed., 2003.

[87] L. Sekanina, "Towards evolvable IP cores for FPGAs," in *Evolvable Hardware, 2003. Proceedings. NASA/DoD Conference*, 2003, pp. 145-154.

[88] J. M. Moreno, E. Sanchez, and J. Cabestany, "An in-system routing strategy for evolvable hardware programmable platforms," in *Evolvable Hardware, Proceedings. The Third NASA/DoD Workshop*, 2001, pp. 157-166.

[89] J. Wang, C. H. Piao, and C. H. Lee, "FPGA Implementation of Evolvable Characters Recognizer with Self-adaptive Mutation Rates," in *International Conference on Adaptive and Natural Computing Algorithms ICANNGA'07*, Warsaw, Poland, 2007, pp. 286-295.

[90] P. C. Haddow, G. Tufte, and P. van Remortel, "Shrinking the genotype: L-systems for Evolvable Hardware," *Lecture Notes in Computer Science,* vol. 2210, 2001.

[91] C. G. Schaefer, Jr., "Morphogenesis of path plan sequences through genetic synthesis of L-system productions," in *Evolutionary Computation, CEC 99. Proceedings of the 1999 Congress*, 1999, p. 365 Vol. 1.

[92] I. Kajitani, T. Hoshino, M. Iwata, and T. Higuchi, "Variable length chromosome GA for evolvable hardware," in *Evolutionary Computation, Proceedings of IEEE International Conference*, 1996, pp. 443-447.

[93] M. Iwata, I. Kajitani, H. Yamada, H. Iba, and T. Higuchi, "A Pattern Recognition System Using Evolvable Hardware," *Lecture Notes In Computer Science,* vol. 1141, pp. 761-770, 1996.

[94] A. Thompson, I. Harvey, and P. Husbands, "The natural way to evolve hardware," in *Circuits and Systems, ISCAS'96., 'Connecting the World', 1996 IEEE International Symposium*, 1996, pp. 37-40 vol.4.

[95] J. C. Gallagher, S. Vigraham, and G. Kramer, "A family of compact genetic algorithms for intrinsic evolvable hardware," *Evolutionary Computation, IEEE Transactions on,* vol. 8, pp. 111-126, 2004.

[96] J. Lee and J. Sitte, *Designing a Morphogenetic System for Evolvable Hardware*, 3339 ed., 2004.

[97] T. G. W. Gordon and P. J. Bentley, "Towards development in evolvable hardware," in *Evolvable Hardware, Proceedings. NASA/DoD Conference*, 2002, pp. 241-250.

[98] J. Torresen, "Scalable evolvable hardware applied to road image recognition," in *Evolvable Hardware, Proceedings. The Second NASA/DoD Workshop* 2000, pp. 245-252.

[99] J. Torresen, "An Evolvable Hardware Tutorial," *Lecture Notes in Computer Science,* pp. 821-830, 2004.

[100] B. Shackleford, G. Snider, R. J. Carter, E. Okushi, M. Yasuda, S. K., and H. Yasuura, "A High-Performance, Pipelined, FPGA-Based Genetic Algorithm Machine " *Genetic Programming and Evolvable Machines,* vol. 2, pp. 33-60, 2004.

[101] T. Maruyama, T. Funatsu, and T. Hoshino, *A Field-Programmable Gate-Array System for Evolutionary Computation*, 1482 ed., 1998.

[102] Z. Yang, S. L. Smith, and A. M. Tyrrell, "Digital circuit design using intrinsic evolvable hardware," in *Evolvable Hardware, Proceedings. 2004 NASA/DoD Conference*, 2004, pp. 55-62.

[103] A. M. Tyrrell, R. A. Krohling, and Y. Zhou, "Evolutionary algorithm for the promotion of evolvable hardware," *Computers and Digital Techniques, IEE Proceedings,* vol. 151, pp. 267-275, 2004.

[104] K. C. Tan, C. M. Chew, K. K. Tan, L. F. Wang, and Y. J. Chen, "Autonomous robot navigation via intrinsic evolution," in *Evolutionary Computation,CEC '02. Proceedings of the 2002 Congress*, 2002, pp. 1272-1277.

[105] A. M. M.Okura, H.Ikeda, and K.Murase, "Artifical evolution of FPGA thta controls a Miniature Mobile Robot Khepera," in *SICE Annual Conference in Fukui*, Fukui University, Japan, 2003.

[106] H. H. Lund, J. Hallam, and W.-P. Lee, "Evolving robot morphology," in *Evolutionary Computation, 1997., IEEE International Conference*, 1997, pp. 197-202.

[107] P. Lysaght, J. Stockwood, J. Law, and D. Girma, *Artificial neural network implementation on a fine-grained FPGA*, 849 ed., 1994.

[108] M. Marchesi, G. Orlandi, F. Piazza, and A. Uncini, "Fast neural networks without multipliers," *Neural Networks, IEEE Transactions on,* vol. 4, pp. 53-62, 1993.

[109] B. Prieto, J. de Lope, and D. o. Maravall, *Reconfigurable Hardware Implementation of Neural Networks for Humanoid Locomotion*, 3562 ed., 2005.

[110] S. L. Bade and B. L. Hutchings, "FPGA-based stochastic neural networks-implementation," in *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, 1994, pp. 189-198.

[111] M. A. Hannan Bin Azhar and K. R. Dimond, "Design of an FPGA based adaptive neural controller for intelligent robot navigation," in *Digital System Design, Proceedings. Euromicro Symposium*, 2002, pp. 283-290.

[112] F. Mondada and D. Floreano, "Evolution of neural control structures: some experiments on mobile robots," *Laboratory of Microcomputing (LAMI), Swiss Federal Institute of Technology, Lausanne, Switzerland*.

[113] A. L. Nelson, E. Grant, and G. Lee, "Developing evolutionary neural controllers for teams of mobile robots playing a complex game," in *Information Reuse and Integration, IRI IEEE International Conference*, 2003, pp. 212-218.

[114] K.-J. Kim and S.-B. Cho, "Dynamic selection of evolved neural controllers for higher behaviors of mobile robot," in *Computational Intelligence in Robotics and Automation, 2001. Proceedings 2001 IEEE International Symposium on*, 2001, pp. 467-472.

[115] E. Tuci, M. Quinn, and I. Harvey, "Evolving fixed-weight networks for learning robots," in *Evolutionary Computation, CEC'02. Proceedings of the Congress*, 2002, pp. 1970-1975.

[116] J. Fernandes-Leon, M. Tosini, and G. G. Acosta, "Evolutionary reactive behavior for mobile robots navigation," in *Cybernetics and Intelligent Systems, IEEE Conference* 2004, pp. 532-537 vol.1.

[117] V. Abhishek, A. Mukerjee, and H. Karnick, "Artificial ontogenesis of controllers for robotic behavior using VLG GA," in *Systems, Man and Cybernetics, IEEE International Conference*, 2003, pp. 3376-3383 vol.4.

[118] A. Berlanga, P. Isasi, A. Sanchis, and J. M. Molina, "Neural networks robot controller trained with evolution strategies," in *Evolutionary Computation, 1999. CEC 99. Proceedings of the Congress* 1999, p. 419 Vol. 1.

[119] J. C. Gallagher, S. K. Boddhu, and S. Vigraham, "A reconfigurable continuous time recurrent neural network for evolvable hardware applications," in *Evolutionary Computation, The IEEE Congress*, 2005, pp. 2461-2468

[120] P. Rocke, J. Maher, and F. Morgan, "Platform for Intrinsic Evolution of Analogue Neural Networks," in *Reconfigurable Computing and FPGAs, 2005. ReConFig 2005. International Conference on*, 2005, p. 11.

[121] R. Manjunath and K. S. Gurumurthy, "Artificial neural networks as building blocks of mixed signal FPGA," in *Field-Programmable Technology (FPT), Proceedings. IEEE International Conference*, 2003, pp. 375-378.

[122] D. Roggen, S. Hofmann, Y. Thoma, and D. Floreano, "Hardware spiking neural network with run-time reconfigurable connectivity in an autonomous robot," in *Evolvable Hardware, Proceedings. NASA/DoD Conference*, 2003, pp. 189-198.

[123] M. A. Hannan Bin Azhar and K. R. Dimond, *Hardware Implementation of a Genetic Controller and Effects of Training on Evolution*, 2606 ed., 2003.

[124] J. F. M. Amaral, C. Santini, R. Tanscheit, M. Vellasco, and M. Pacheco, "Towards evolvable analog artificial neural networks controllers," in *Evolvable Hardware, Proceedings. 2004 NASA/DoD Conference*, 2004, pp. 46-52.

[125] K. Sung Hoe, P. Chongkug, and F. Harashima, "A self-organized fuzzy controller for wheeled mobile robot using an evolutionary algorithm," *Industrial Electronics, IEEE Transactions,* vol. 48, pp. 467-474, 2001.

[126] I.-K. Jeong and J.-J. Lee, "Evolving fuzzy logic controllers for multiple mobile robots solving a continuous pursuit problem," in *Fuzzy Systems Conference Proceedings, FUZZ-IEEE '99. IEEE International*, 1999, pp. 685-690 vol.2.

[127] L. Doitsidis and N. C. Tsourveloudis, "An Empirical Study for Fitness Function Selection in Fuzzy Logic Controllers for Mobile Robot Navigation," in *IEEE Industrial Electronics, IECON 32nd Annual Conference*, 2006, pp. 3868-3873.

[128] D. Gu and H. Hu, "Evolving Fuzzy Logic Controllers for Sony Legged Robots RoboCup 2001: Robot Soccer World Cup V." vol. 2377, A. Birk, S. Coradeschi, and S. Tadokoro, Eds., ed: Springer Berlin / Heidelberg, 2002, pp. 5-11.

[129] D. Gu, H. Hu, J. Reynolds, and E. Tsang, "GA-based learning in behaviour based robotics," in *Computational Intelligence in Robotics and Automation, Proceedings IEEE International Symposium* 2003, pp. 1521-1526 vol.3.

[130] T.-H. s. Li, S.-J. Chang , and Y.-X. Chen "Implementation of human-like driving skills by autonomous fuzzy behavior control on an FPGA-based car-like mobile robot," *Industrial Electronics, IEEE Transactions on,* vol. 50, pp. 867-880, 2003.

[131] S. Wu, Q. Li, E. Zhu, J. Xie, and G. Zhichao, "Fuzzy controller of pipeline robot navigation optimized by genetic algorithm," in *Control and Decision Conference, CCDC Chinese*, 2008, pp. 904-908.

[132] G. Chronis, J. Keller, and M. Skubic, "Learning fuzzy rules by evolution for mobile agent control," in *Computational Intelligence in Robotics and Automation,CIRA'99. Proceedings IEEE International Symposium*, 1999, pp. 70-76.

[133] M. S. Islam, M. S. Bhuyan, M. A. Azim, L. K. Teng, and M. Othman, "Hardware Implementation of Traffic Controller using Fuzzy Expert System," in *Evolving Fuzzy Systems, International Symposium*, 2006, pp. 325-330.

[134] R. O. Ambrose, R. T. Savely, S. M. Goza, P. Strawser, M. A. Diftler, I. Spain, and N. Radford, "Mobile manipulation using NASA's Robonaut," in *Robotics and Automation, Proceedings. ICRA '04. IEEE International Conference* 2004, pp. 2104-2109 Vol.2.

[135] B. Browning, P. E. Rybski, J. Searock, and M. M. Veloso, "Development of a soccer-playing dynamically-balancing mobile robot," in *IEEE International*

Conference on Robotics and Automation, New Orleans, LA, USA, 2004, pp. 1752-7.

[136] F. Grasser, A. D'Arrigo, S. Colombi, and A. C. Rufer, "JOE: a mobile, inverted pendulum," *Industrial Electronics, IEEE Transactions on,* vol. 49, pp. 107-114, 2002.

[137] K. K. Noh, J. G. Kim, and U. Y. Huh, "Stability experiment of a biped walking robot with inverted pendulum," in *Industrial Electronics Society, 2004. IECON 2004. 30th Annual Conference of IEEE*, 2004, pp. 2475-2479 Vol. 3.

[138] Y. Kim, S. Kim, and Y. Kwak, "Dynamic Analysis of a Nonholonomic Two-Wheeled Inverted Pendulum Robot," *Journal of Intelligent and Robotic Systems,* vol. 44, pp. 25-46, 2005.

[139] C. Huang, W. Wang, and C. Chiu, "Design and Implementation of Fuzzy Control on a Two-Wheel Inverted Pendulum," *Industrial Electronics, IEEE Transactions on,* vol. PP, pp. 1-1.

[140] C. W. Anderson, "Learning to control an inverted pendulum using neural networks," *Control Systems Magazine, IEEE,* vol. 9, pp. 31-37, 1989.

[141] E. Pasero and M. Perri, "Hw-Sw codesign of a flexible neural controller through a FPGA-based neural network programmed in VHDL," in *Neural Networks, Proceedings. IEEE International Joint Conference* 2004, pp. 3161-3165 vol.4.

[142] S. Jung and S. S. Kim, "Control Experiment of a Wheel-Driven Mobile Inverted Pendulum Using Neural Network," *Control Systems Technology, IEEE Transactions on,* vol. 16, pp. 297-303, 2008.

[143] J. N. Seok, G. H. Lee, H. J. Choi, and S. Jung, "Robust control of a mobile inverted pendulum robot using a RBF neural network controller," in *Robotics and Biomimetics, ROBIO IEEE International Conference on*, 2008, pp. 1932-1937.

[144] J. S. Noh, G. H. Lee, and S. Jung, "Motion control of a mobile pendulum system using neural network," in *Advanced Motion Control, AMC '08. 10th IEEE International Workshop* 2008, pp. 450-454.

[145] M. Obika, K. Kawada, S. Fujisawa, T. Yamamoto, and Y. Suita, "The creation of the motion pattern attended with emergence using evolutional computation," in *SICE Annual Conference*, 2003, pp. 3283-3287

[146] F. Hoffmann, "Evolutionary algorithms for fuzzy control system design," *Proceedings of the IEEE,* vol. 89, pp. 1318-1333, 2001.

[147] M. Y. Shieh, C. W. Huang, and T. H. S. Li, "A GA-based Sugeno-type fuzzy logic controller for the cart-pole system," in *Industrial Electronics, Control and*

Instrumentation, IECON97. 23rd International Conference, 1997, pp. 1028-1033 vol.3.

[148] K. Min-Soeng and L. Ju-Jang, "Constructing a fuzzy logic controller using evolutionary Q-learning," in *Industrial Electronics Society, . IECON 26th Annual Conffference of the IEEE*, 2000, pp. 1785-1790 vol.3.

[149] Z. Xuerui, T. Gang, L. Junho, and P. Allaire, "Control implementation for a balance beam with magnetic bearings," in *American Control Conference,. Proceedings of the* 2000, pp. 3069-3070 vol.5.

[150] C. Ka and L. Nan, "A ball balancing demonstration of optimal and disturbance-accomodating control," *Control Systems Magazine, IEEE,* vol. 7, pp. 54-57, 1987.

[151] F. Gordillo, F. Gómez-Estern, R. Ortega, and J. Aracil, "On the ball and beam problem: regulation with guaranteed transient performance and tracking periodic orbits," presented at the Proc of the International Symposium on Mathematical Theory of Networks and Systems, University of Notre Dame, IN, USA, 2002.

[152] E. P. Dadios, R. Baylon, R. De Guzman, A. Florentino, R. M. Lee, and Z. Zulueta, "Vision guided ball-beam balancing system using fuzzy logic," in *Industrial Electronics Society, IECON . 26th Annual Conffference of the IEEE*, 2000, pp. 1973-1978 vol.3.

[153] J. Iqbal, M. A. Khan, S. Tarar, M. Khan, and Z. Sabahat, "Implementing ball balancing beam using digital image processing and fuzzy logic," in *Electrical and Computer Engineering, Canadian Conference* 2005, pp. 2241-2244.

[154] K. C. Ng and M. M. Trivedi, "Neural integrated fuzzy controller (NiF-T) and real-time implementation of a ball balancing beam (BBB)," in *Robotics and Automation, Proceedings IEEE International Conference on*, 1996, pp. 1590-1595 vol.2.

[155] P. H. Eaton, D. V. Prokhorov, and D. C. Wunsch, II, "Neurocontroller alternatives for fuzzy ball-and-beam systems with nonuniform nonlinear friction," *Neural Networks, IEEE Transactions,* vol. 11, pp. 423-435, 2000.

[156] H. Benbrahim, J. S. Doleac, J. A. Franklin, and O. G. Selfridge, "Real-time learning: a ball on a beam," in *Neural Networks, IJCNN, International Joint Conference on*, 1992, pp. 98-103 vol.1.

[157] A. G. B. Tettamanzi, "An evolutionary algorithm for fuzzy controller synthesis and optimization," in *Systems, Man and Cybernetics, Intelligent Systems for the 21st Century, IEEE International Conference on*, 1995, pp. 4021-4026

[158] Z. Yi and Y. Xiuxia, "Design for beam-balanced system controller based on chaos genetic algorithm," in *Information Acquisition, Proceedings. International Conference* 2004, pp. 448-451.

[159] A. L. Nelson and E. Grant, "Developmental Analysis in Evolutionary Robotics," in *Adaptive and Learning Systems,IEEE Mountain Workshop* 2006, pp. 201-206.

[160] *IEEE Standards: IEE Standard Verilog Hardware Description Language*, IEEE Std 1364-2001 ed.: The Institute of Electrical and Electronics Engineers, Inc., 2001.

Appendix A Published Papers

A GA based Controller for a Mobile Inverted Pendulum

A GA based Controller for a Mobile Inverted Pendulum

Mark Beckerleg, John Collins
Engineering Research Institute
Auckland University of Technology, Auckland, New Zealand
m.beckerleg@aut.ac.nz

Abstract

Previous controllers for a two wheeled mobile inverted pendulum have used standard PID techniques. This paper presents a novel method which uses genetic algorithms to evolve the control system for a computer simulation of a mobile inverted pendulum. The heart of the controller is a lookup table that relates the current pendulum's angle and angular velocity to the motor direction and torque settings required to maintain balance of the pendulum. The lookup table is modified by a genetic algorithm using tournament selection, two-point crossover and creeping mutation with a 1% possibility of mutation occurring. It is shown that controllers capable of balancing the pendulum for time periods greater than three minutes could evolve within 200 generations.

Keywords: Evolutionary Computation, Genetic Algorithm, Mobile Inverted Pendulum.

1 Introduction

Mobile inverted pendulums have moved from university research to commercial reality with the advent of the Segeway. With the commercialization of the pendulum and model applications such as remotely controlled helicopters, gyroscopes and accelerometers have come down to affordable levels enabling universities and students to use these in project applications. The use of these devices and the pendulum model can also be modified for other fields such as bipedal locomotion [1] or game playing such as soccer [2].

Mathematical models describing the behaviour of the pendulum have been developed enabling the construction of PID controllers which provide excellent balancing performance [3, 4]. Other researchers have investigated fuzzy logic [5-7] and neural network [8, 9] controllers to maintain the pendulum balance.

This paper investigates the techniques of evolutionary computation and genetic algorithms to evolve the controller for the pendulum. Evolutionary processes have been extensively researched for robotic applications such as robotic navigation using evolutionary computation, evolvable hardware and evolutionary artificial neural networks. Genetic algorithms have not previously been used to control a mobile inverted pendulum.

The main difficulty in the field of evolutionary robotics is the large search space and time required to evolve a controller. Several methods have been used to try to overcome this by either reducing the chromosome and thus the search space [10], or by using subsumption architecture where individual behaviours of the robot are evolved independently of each other [11, 12]. In this application we use subsumption where the first behaviour to evolve is initially balancing. Future behaviours such as navigation or autonomy can be independently evolved.

The Auckland University of Technology has built a mobile inverted pendulum as part of a research project for evolutionary computation and evolvable hardware. The pendulum has been balanced with a controller using standard PID control software based on previous research. The hardware is comprised of two parts; a motherboard which has the physical components such as gyroscopes, accelerometers, motor drivers and wheel sensors, required to balance the pendulum, and a daughter board which contains the processor used for control.

Three types of daughter boards have been created with three different processor cores, an Atmel 8 bit microcontroller, an ARM 32 bit processor and an Altera Cyclone II FPGA. The daughter boards have been created as a medium to provide research into evolutionary computation and several robotic platforms have been developed which interface to these boards. This will allow research into evolvable artificial neural networks, evolvable fuzzy logic and evolvable hardware controllers. The first stage of the research is to create a simulation of the pendulum to allow testing of genetic algorithms.

2 Mathematical Modelling

The pendulum is an example of a non-holonomic robot with three degrees of freedom (DOF), two planar motions and one tilt-angular motion, but with direct control in only the planar motions. Thus the control of the planar motion must work in such a way as to control the tilt/angular motion. As shown in figure 1, the pendulum can rotate around the z axis (tilt), this is described by its angle θ_p and its angular

velocity ω_p. The pendulum can move on its x axis described by its position x and its velocity υ. The pendulum can also rotate around the y axis described by the angle δ and angular velocity δ'

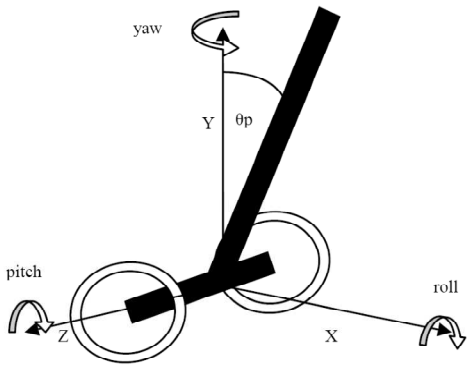


Figure 1 Physical Model of a Pendulum

For this simulation the two wheels are driven together so there will be no yaw, and the pendulum is on a flat surface so there will be no roll. The important parameters are the pendulum's angle, angular velocity and the linear displacement along the x-axis.

It can be shown that the equations (1) and (2) describing the motion of the system are

$$\left(M_p + 2M_w + \frac{2I_w}{r^2}\right)\ddot{x} + M_p l\cos 0_p \ddot{0}_p = \frac{2T}{r} + M_p l\dot{\theta}_p{}^2 \sin\theta_p \quad (1)$$

$$M_p l\cos\theta_p \ddot{x} + I_p\ddot{\theta}_p = M_p gl\sin\theta_p - 2T \quad (2)$$

The computer simulation solves equations in quantized time intervals to determine the pendulums current acceleration $\ddot{x}$, horizontal displacement $\dot{x}$, angle θ_p and angular velocity $\dot{\theta}_p$

It is assumed that the motor torque T varies linearly with applied motor voltage and motor speed.

3 Genetic Algorithm

3.1 Chromosome

In evolutionary computation terms the chromosome is a possible solution or solutions to a problem. A single solution is termed an individual and a group of solutions is termed a population. In regard to the pendulum the solution is the required motor direction and torque for a given angle and angular velocity. The motor driver for the pendulum is a H-bridge driver which is controlled by an eight bit number. The number is a linear representation of the motor direction and torque with 0 representing the maximum reverse torque, 125 motor stopped and 250 the maximum forward torque. The number step size is 25 allowing eleven possible torque settings.

The chromosome is a two dimensional lookup table which converts the angle and angular velocity of the pendulum to the required motor direction and torque. The x-axis represents the pendulum's angle ranging ±18 degrees from vertical. The angle step size is 3 degrees. The y-axis represents the pendulum's angular velocity of the pendulum ranging ±30 degrees/second. The angular velocity step size is 5 degrees/second.

The population size is 100 with the initial population randomly generated.

3.2 Reproduction and Selection

Chromosome reproduction uses a two point crossover scheme where two points within the chromosome are randomly chosen and the gene code within these two points is swapped between the two parents. Creeping mutation is also used, where the gene is replaced with a value within 10% of the non-mutated value. The probability of a mutation occurring is 1% and every mutation will randomly change 10 genes within the individual's chromosome.

Tournament selection is used to select the best individuals. This process selects a subgroup of individuals from the population and only the fittest individual is retained. For this example the subgroup size is two which has poor selection intensity but maintains the widest gene pool. An added advantage is that tournament selection requires no sorting of the population

3.3 Fitness

The fitness is determined by the length of time that the pendulum remains within a vertical range of $\pm 18^0$ and remains within ± 0.5 meters of its start position.

4 Simulation

The simulation shown in figure 2 has a diagrammatic representation of the pendulum and numerical displays of the initial and current pendulum values of horizontal displacement, horizontal velocity, angle, angular velocity, individual balancing time and motor torque settings. The pendulum starting angle, current generation, individual, average fitness and maximum fitness are also displayed. These parameters are also saved to memory for future analysis. The simulation can be set to run in real time where the motion of the pendulum can be observed or it can be speeded up allowing faster evolution but the pendulum can not be observed.

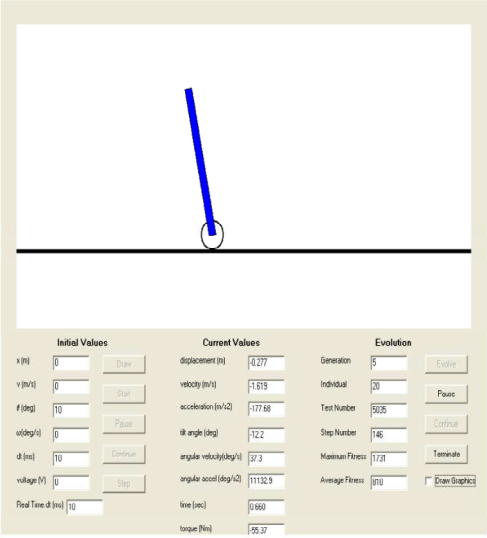


Figure 2 Simulation GUI

5 Results

5.1 Simulation and GA modifications

When the simulation was first run it was found that under certain conditions the controller would stop evolving. To prevent this both the simulation and the fitness criteria needed to be modified. These modifications are described next.

5.1.1 Variation of the start angle

It was found only a limited part of the controller array would evolve. Under investigation it was determined that the pendulum would learn to move from its initial offset starting angle and then maintain its balance. However the pendulum never evolved how to balance from other start conditions. To overcome this, the same individual is tested over a range of starting positions. The fitness was modified to accumulate the total time that the pendulum remained upright.

5.1.2 Horizontal drift

The requirement is for the pendulum to maintain a balance within a limited horizontal displacement. However it was found that the pendulum would find a balance point with a vertical offset giving a constant horizontal motion. In order to penalize this characteristic, the individual under going fitness evaluation was terminated when the pendulum moved ± 0.5 meters from the initial horizontal starting position. Thus the fitness for each individual is determined by both its ability to balance the pendulum and to remain stationary.

5.1.3 Individual not leaving a test

It was found that individual solutions were capable of kicking the pendulum upright from its initial starting position and then stabilising it within the ±0.5 meter range. This is a good result for one individual at one starting point, however the fitness evaluation would never end and other start angles for that individual and other individuals in the population would never evolve. To overcome this all individuals are terminated after a 5 minute interval.

5.2 Partial evolution

It was found that some parameters of the array related to a high pendulum angle with high angular velocities did not evolve to the motor torque value expected from the mathematical model, i.e. moving to a maximum forward or reverse torque. This is because at these positions the pendulum is not able to maintain balance, even with the motor at maximum torque. Therefore the resulting fitness will allow suitable evolution of those parts.

5.3 Fitness

Five simulations were run for a maximum of 200 generations and the best and average fitness of each simulation was recorded. A typical result for evolution using tournament selection is shown in figure 3 with an initial large gap between the best and average fitness, with a gradual improvement of the average fitness with each generation. The best individual would have been expected to reach 300 seconds, but instead the best individual plateaus at 150 seconds with only small improvements after that. It is likely that the evolution was moving into local maxima. This possibly could be avoided by increasing the mutation rate.

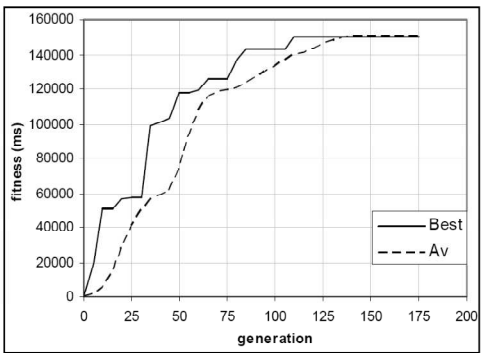


Figure 3 Average and Best Fitness

Figures 4 to 5 graph the recorded best and average fitness of the five simulations. It can be seen that the best fitness stays constant for several generations and then increases in a large step when a better individual has been reproduced. The average fitness of the population has a gradual increase every generation as the individuals slowly improve towards the best individual.

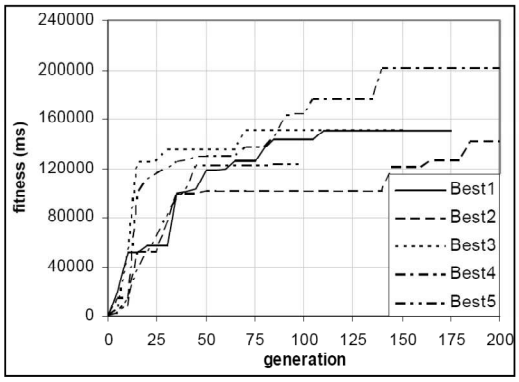


Figure 4 Best Fitness

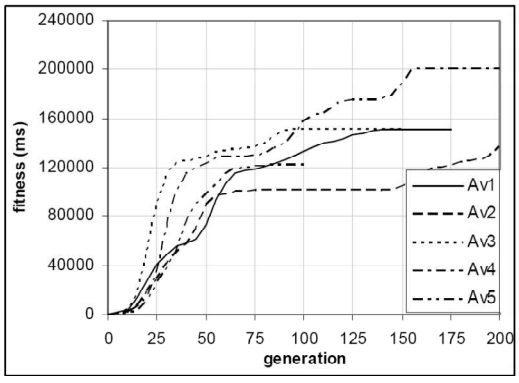


Figure 5 Average Fitness

6 Conclusions

This paper has described a method where a simulation of a mobile inverted pendulum controller can be evolved to a point where an acceptable level of balance is achieved. It was found that the GA produced a controller capable of balancing the pendulum for two minutes within 70 generations.

7 Future research

The future aims of this project are to evolve the pendulum controller using both evolutionary computation and evolvable hardware. This can be researched using the three daughter boards with a range of processors and FPGAs. The AUT mobile pendulum has been built with trainer wheels which can be set so that it will start at a $\pm 18^0$ position.

A background software routine will be designed to reset the pendulum back to a starting position and angle. It will allow individuals to be tested, and take back control of the pendulum if it moves outside permitted limits, allowing the pendulum controller to evolve with out human interaction. The background software will either perform the genetic algorithms if enough processing power is available, or it will transmit the data to a PC where the genetic algorithm can be performed.

8 Nomenclature

The following variables represent the pendulums characteristics that the model is dependant on.

g = acceleration of gravity
I_p = pendulums moment of inertia around the z axis
l =distance to pendulum centre of mass from the z axis
I_w = moment of inertia of wheel around the z axis
M_p = mass of the pendulum
M_w = mass of the wheel
r = radius of the wheel
T = motor torque
x = horizontal displacement of the wheels
θ_p = angle of pendulum relative to the vertical axis

9 References

[1] K.-K. Noh, J.-G. Kim, and U.-Y. Huh, "Stability experiment of a biped walking robot with inverted pendulum," presented at Industrial Electronics Society, 2004. IECON 2004. 30th Annual Conference of IEEE, 2004.

[2] B. Browning, P. E. Rybski, J. Searock, and M. M. Veloso, "Development of a soccer-playing dynamically-balancing mobile robot," presented at 2004 IEEE International Conference on Robotics and Automation, 26 April-1 May 2004, New Orleans, LA, USA, 2004.

[3] F. Grasser, A. D'Arrigo, S. Colombi, and A. C. Rufer, "JOE: a mobile, inverted pendulum," *Industrial Electronics, IEEE Transactions on*, vol. 49, pp. 107-114, 2002.

[4] Y. Kim, S. Kim, and Y. Kwak, "Dynamic Analysis of a Nonholonomic Two-Wheeled Inverted Pendulum Robot," *Journal of Intelligent and Robotic Systems*, vol. 44, pp. 25-46, 2005.

[5] D. Xue-ming, Z. Pei-ren, Y. Xing-ming, and X. Yong-ming, "The application of hierarchical fuzzy control for two-wheel mobile inverted pendulum," *Electric Machines and Control*, vol. 9, pp. 372-5, 2005.

[6] S. S. Ge, Y. K. Loi, and C.-Y. Su, "Fuzzy logic control of a pole balancing vehicle," presented at Proceedings of the 3rd International Conference on Industrial

Automation, 7-9 June 1999, Montreal, Que., Canada, 1999.

[7] L. Ojeda, M. Raju, and J. Borenstein, "FLEXnav: a fuzzy logic expert dead-reckoning system for the Segway RMP," presented at Unmanned Ground Vehicle Technology VI, 13-15 April 2004

Proceedings of the SPIE - The International Society for Optical Engineering, Orlando, FL, USA, 2004.

[8] S.-s. Kim, T. I. Kim, K. S. Jang, S. Jung, S. S. Kim, T. I. Kim, K. S. Jang, and S. Jung, "Control experiment of a wheeled drive mobile pendulum using neural network," presented at IECON 2004 - 30th Annual Conference of IEEE Industrial Electronics Society, 2-6 Nov. 2004, Busan, South Korea, 2004.

[9] T.-C. Chen, T.-J. Ren, and C.-H. Yu, "Motion control of a two-wheel power aid mobile," presented at SICE Annual Conference 2005, Aug 8-10 2005, Okayama, Japan, 2005.

[10] I. Harvey, *Artificial Evolution: A Continuing SAGA*, 2217 ed, 2001.

[11] R. A. Brooks, "A robot that walks; emergent behaviors from a carefully evolved network," presented at Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on, 1989.

[12] D. Cliff, I. Harvey, and P. Husbands, "Evolutionary robotics," presented at Design and Development of Autonomous Agents, IEE Colloquium on, 1995.

An Analysis of the Chromosome Generated by a Genetic Algorithm Used to Create a Controller for a Mobile Inverted Pendulum

Mark Beckerleg, John Collins

Engineering Research Institute
Auckland University of Technology, Auckland, New Zealand
m.beckerleg@aut.ac.nz, jcollins@aut.ac.nz

Abstract

This paper presents a novel method which uses a genetic algorithm to evolve the control system of a computer simulated mobile inverted pendulum. During the evolutionary process the best and average fitness, the chromosome and pendulum motion is recorded and analysed. The chromosome is a two dimensional lookup table that relates the current pendulum's angle and angular velocity to the motor direction and torque settings required to maintain balance of the pendulum. The lookup table is modified by a genetic algorithm using tournament selection, two-point crossover and a mutation rate of 2%. After two hundred generations the evolved chromosome is capable of balancing the pendulum for more than two hundred seconds.

1 Introduction

Mobile inverted pendulums have moved from university research to commercial reality with the advent of the Segway. The reduction in costs of gyroscopes and accelerometers has enabled universities and students to use these in project applications. The use of these devices and the pendulum model can also be modified for other fields such as bipedal locomotion [1] or game playing such as soccer [2].

Mathematical models describing the behaviour of the pendulum have been developed enabling the construction of PID controllers which provide excellent balancing performance [3, 4]. Other researchers have investi-

gated fuzzy logic [5-7] and neural network [8, 9] controllers to maintain the pendulum balance. Genetic algorithms have not previously been used to control a mobile inverted pendulum.

This paper is an extension of a paper first published in ICARA 2006 [10] which investigates the techniques and associated problems of evolutionary computation and genetic algorithms to evolve the controller for the pendulum.

The main difficulties in the field of evolutionary robotics are: (1) initial chromosomes are destructive to the robot and its environment; (2) initial chromosome populations have very little selective pressure (bootstrap problem [11]); (3) robotic tasks are complex creating a large search space and time required to evolve a controller. Robotic simulation is used to overcome the first two problems. The latter can be diminished by either reducing the chromosome size and thus the search space [12], or by using subsumption architecture where individual behaviours of the robot are evolved independently of each other before being combined together [13, 14]. In this application we use subsumption architecture in which the first behaviour to evolve is balancing. Future behaviours such as navigation or autonomy can then be independently evolved.

The Auckland University of Technology has constructed a mobile inverted pendulum for this research project which is capable of balancing using standard PID control software. The hardware is capable of supporting either an Atmel 8 bit microcontroller, an ARM 32 bit processor or an Altera Cyclone II FPGA allowing research into evolutionary computation, evolvable artificial neural networks, evolvable fuzzy logic and evolvable hardware. The first stage of the research is to create a simulation of the pendulum to allow testing of genetic algorithms.

2 Mathematical Modeling

The pendulum is a non-holonomic robot with three degrees of freedom (DOF), two planar motions and one tilt-angular motion, but with direct control in only the planar motions. Thus the control of the planar motion must work in such a way as to control the tilt/angular motion. As shown in Figure 1, the pendulum can rotate around the z axis (tilt), this is described by its angle θp and its angular velocity ωp. The pendulum can move on its x axis described by its position x and its velocity υ. The pendulum can also rotate around the y axis described by the angle Φ and angular velocity Φ'.

To simplify the evolutionary process, the pendulum will be constrained to only one planar axis x by driving the two wheels together so there will be no yaw, and the pendulum is on a flat surface so there will be no roll.

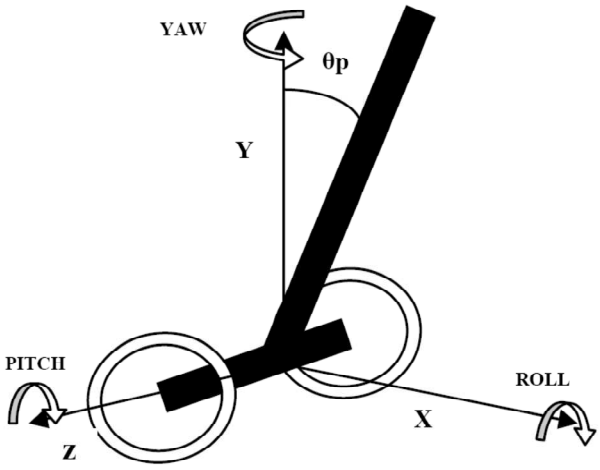


Fig. 1 Physical Model of a Pendulum

The important parameters are the pendulum's angle, angular velocity and the linear displacement along the x-axis. It can be shown that the motion of the pendulum can be described by equations (1) and (2).

$$\left(M_p + 2M_w + \frac{2I_w}{r^2}\right)\ddot{x} + M_p l \cos\theta_p \ddot{\theta}_p = \frac{2T}{r} + M_p l \dot{\theta}_p{}^2 \sin\theta_p \quad (1)$$

$$M_p l \cos\theta_p \ddot{x} + I_p \ddot{\theta}_p = M_p g l \sin\theta_p - 2T \quad (2)$$

The computer simulation solves these equations in quantized time intervals to determine the pendulum's current horizontal velocity, horizontal displacement, angle and angular velocity.

3 Genetic Algorithm

A chromosome is a possible solution to a problem. In regard to the pendulum, the chromosome is the required motor direction and torque for a given angle and angular velocity. The motor driver for the pendulum is an H-bridge driver controlled by an eight bit number. The number is a linear representation of the motor direction and torque, with 0 representing the maximum reverse torque, 125 is the motor stopped and 250 is the maximum forward torque. The step size is 25 allowing eleven possible torque settings.

The chromosome is a two dimensional lookup table which relates the angle and angular velocity of the pendulum to the required motor direction and torque. The rows represent the pendulum's angle ranging ±18 degrees from vertical with a step size of 3 degrees. The columns represent the pen-

dulum's angular velocity ranging ±30 degrees/second with a step size of 5 degrees/second. The population size is 100 with the initial population randomly generated

Chromosome reproduction creates two offspring per pair of parents, using a two point crossover scheme where two points within the chromosome are randomly chosen and the gene code within these two points is swapped between the two parents. To maintain population diversity two percent of the population is mutated, where each mutated chromosome has ten genes randomly altered.

The selection process used is tournament selection in which a subgroup of individuals is selected from the population and only the fittest individual within the subgroup is retained. The selection pressure is increased as the subgroup size increases because fewer individuals are being kept in each generation, however the gene pool will rapidly decrease with a high selection pressure, increasing the possibility of the evolutionary process becoming stuck in local maxima. For this experiment the subgroup size is two, which has poor selection intensity but maintains the widest gene pool.

The fitness is determined by the length of time that the pendulum remains within a vertical range of ±18 degrees and remains within ± 0.5 meters of its start position of 18 degrees with an angular velocity of zero. Each fitness test is stopped after three hundred seconds if the pendulum has not gone outside the maximum vertical range within this time..

4 Simulation

The simulation shown in Figure 2 has a diagrammatic representation of the pendulum and numerical displays of the pendulum's parameters. The pendulum starting angle, current generation, individual, average fitness and maximum fitness are also displayed. These parameters are saved to a file for future analysis. The simulation can be set to run in real time allowing the motion of the pendulum to be observed, or it can be sped up allowing faster evolution but then the pendulum can not be observed.

The best individual's fitness, chromosome and motion were recorded. The motion showed the positions in the array that the individual stepped through as it was tested for fitness. This data allows us to monitor how the chromosome is evolving, to see what parts of the array the individual passed through, where it spent most of its time and what caused the individual to fail the test (angle, horizontal displacement or timeout).

When the simulation was first run it was found that under certain conditions the controller would stop evolving. To prevent this both the simulation and the fitness criteria needed to be modified.

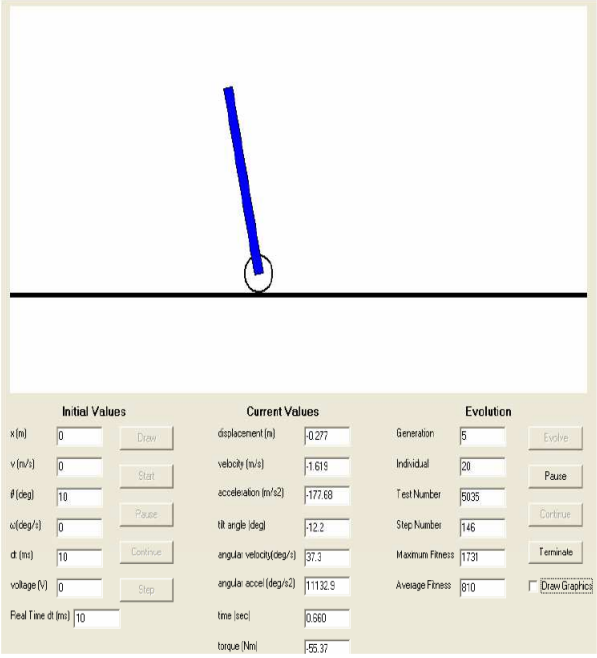


Fig.2 Simulation GUI

It was found only a limited part of the controller array would evolve. This is because the pendulum would learn to balance from its initial offset starting angle. However the pendulum never evolved how to balance from other start conditions. To overcome this, the same individual was tested over a range of starting angles with a starting angular velocity of zero

It was found that the pendulum would maintain a balance point with a constant horizontal motion. This characteristic was penalized by terminating the test when the pendulum moved ± 0.5 meters from the initial horizontal starting position. Thus the fitness for each individual was determined by both its ability to balance the pendulum and to remain stationary.

It was found that some tests would last an indefinite period of time. This is a good result for one individual at one starting point, however the fitness evaluation would never end and the evolution process would stop. To overcome this all individuals are terminated after a 300 second interval.

It was found that the extreme positions of the array did not evolve to the value expected from the mathematical model, i.e. moving to a maximum forward or reverse torque. This is because at these positions the pendulum is not able to maintain balance, even with the motor at maximum torque. Thus parts of the array will not evolve as there are no values that will offer a significant difference in fitness.

It was found that the motor torque was not strong enough to pull the pendulum upright within the ±0.5 meter boundary for the start angles of ±18 and ±15, thus the best possible fitness is about 220 seconds.

5 Results

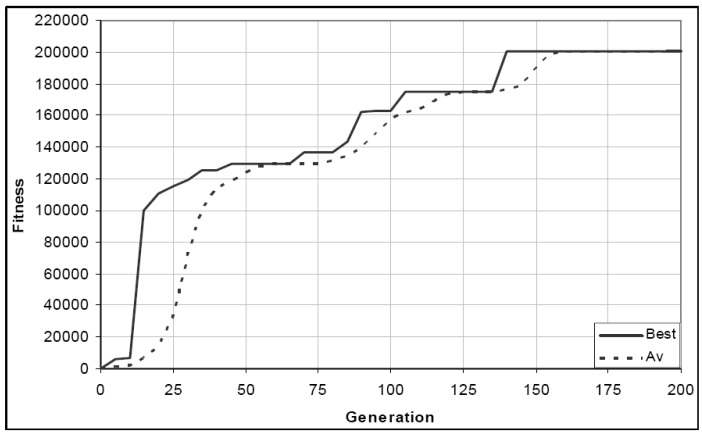


Fig. 3 Average and Best Fitness

Angular Velocity

Angle	- 30	- 25	- 20	- 15	- 10	- 5	0	5	10	15	20	25	30
-18	50	100	225	0	50	175	25	100	50	150	200	50	125
-15	25	250	25	125	0	0	50	125	75	175	250	225	100
-12	175	150	25	25	25	225	75	75	0	175	25	100	75
- 9	250	25	25	25	150	0	100	75	0	200	50	50	100
- 6	25	125	150	75	150	175	75	25	125	0	100	50	0
- 3	100	175	0	150	25	50	0	25	25	0	100	25	225
0	225	100	100	150	25	250	75	150	200	50	100	200	100
3	225	75	75	125	100	250	225	100	225	100	250	0	100
6	200	225	200	175	225	225	0	200	175	100	0	150	25
9	250	150	200	100	175	0	225	125	200	25	175	150	100
12	225	225	175	200	225	125	225	225	150	200	25	225	150
15	250	250	250	225	125	250	200	225	125	125	225	250	250
18	50	250	50	50	75	150	250	250	100	0	175	200	200

Table1 Evolved Chromosomes

A typical result is shown in Figure 3 with an initial large gap between the best and average fitness, and then a convergence between the average and best fitness with each generation. The best fitness increases in steps where as the average fitness gradually improves. An evolved chromosome for a recorded run is shown in Table 1 Each individual was tested over the range of start angles and each step that the pendulum took in the lookup table was recorded.

Initially the simulation was tested with a chromosome from a mathematical model of the pendulum. This model had a linear progression of motor torque dependant on the angle and angular velocity. When this chromosome was tested it performed poorly, failing the test after less than one second. A review of the pendulum's motion showed the reason for

failure was that even though the pendulum moved to an upright position it had horizontal drift outside the ±0.5 meter distance. From this it can be seen that successful chromosomes require torque settings that quickly move the pendulum to a vertical position and then keep it within the horizontal boundary.

An analysis of an evolved chromosome and the sequence that it moves through the chromosome shows the pendulum is quickly brought upright and then jitters around a position so that horizontal drift is eliminated. The pendulum eventually runs to the timeout limit or a small horizontal drift takes the pendulum to the positional limit. The evolved chromosome has a non-linear progression between cells which allows it to become vertical on start up and jitter around an angle of zero degrees.

The path that the pendulum takes through the chromosome is highly convoluted. Some of the cells would seem to have incorrect values according to their angle and angular velocity. However adjacent cells compensate for the incorrect settings. It is this erratic sequence of motor torques that creates the jitter and corresponding horizontal stability.

The best chromosomes from several evolutionary runs were compared and it was found that the chromosomes were quite different even though they had a similar fitness. This is because the initial chromosomes are random and the convoluted pathways that they evolve are unique to each starting individual.

6 Conclusions

This paper has described a method in which a simulation of a mobile inverted pendulum controller can be evolved to a point where an acceptable level of balance is achieved. It was found that the GA produced a controller capable of balancing the pendulum for two hundred seconds within 200 generations. The next stage of research is to move the application from simulation to the AUT mobile inverted pendulum

7 References

[1] K.-K. Noh, J.-G. Kim, and U.-Y. Huh, "Stability experiment of a biped walking robot with inverted pendulum," Industrial Electronics Society, 2004. (IECON) 2004. 30th Annual Conference of IEEE, Busan, Korea, 3, pp2475-2479 (2004).

[2] B. Browning, P. E. Rybski, J. Scarock, and M. M. Veloso, "Development of a soccer-playing dynamically-balancing mobile robot," Proceedings IEEE International Conference on Robotics and Automation, New Orleans, pp 1752-1757 (2004).

[3] F. Grasser, A. D'Arrigo, S. Colombi, and A. C. Rufer, "JOE: a mobile, inverted pendulum," Industrial Electronics, IEEE Transactions on, vol. 49, pp. 107-114, (2002).
[4] Y. Kim, S. Kim, and Y. Kwak, "Dynamic Analysis of a Nonholonomic Two-Wheeled Inverted Pendulum Robot," Journal of Intelligent and Robotic Systems, vol. 44, pp. 25-46, (2005).
[5] X.M. Ding, P.R. Zhang, X.M. Yang, Y.M. Xu, "The application of hierarchical fuzzy control for two-wheel mobile inverted pendulum," Electric Machines and Control, vol. 9, pp. 372-5, (2005).
[6] S. S. Ge, Y. K. Loi, and C.-Y. Su, "Fuzzy logic control of a pole balancing vehicle," Proceedings of the 3rd International Conference on Industrial Automation, Montreal, pp 10-17, (1999).
[7] L. Ojeda, M. Raju, and J. Borenstein, "FLEXnav: a fuzzy logic expert dead-reckoning system for the Segway RMP," Proceedings Unmanned Ground Vehicle Technology VI, International Society for Optical Engineering, Orlando, FL, USA, pp11-23 (2004).
[8] S. S. Kim, T. I. Kim, K. S. Jang, and S. Jung, "Control experiment of a wheeled drive mobile pendulum using neural network," Proceedings 30th Annual Conference of IEEE Industrial Electronics Society (IECON 2004), 2-6 Nov. 2004, Busan, South Korea, pp 2234-2239 (2004).
[9] T.-C. Chen, T.-J. Ren, and C.-H. Yu, "Motion control of a two-wheel power aid mobile," Proceedings SICE Annual Conference 2005, Aug 8-10 2005, Okayama, Japan, (2005).
[10] M. Beckerleg and J. Collins, "A GA Based Controller for a Mobile Inverted Pendulum," 3rd International Conference on Autonomous Robots and Agents (ICARA 2006), pp 123-127, 12-14 December, Palmerston North, New Zealand, (2006).
[11] A. L. Nelson and E. Grant, "Developmental Analysis in Evolutionary Robotics," IEE mountain workshop on Adaptive and Learning Systems, pp 201-206 (2006).
[12] I. Harvey, "Artificial Evolution: A Continuing SAGA," Proceedings Evolutionary Robotics : from intelligent robotics to artificial life (ER 2001), International symposium, Tokyo , JAPAN, 2217, pp 94-109 (2001).
[13] R. A. Brooks, "A robot that walks; emergent behaviours from a carefully evolved network," Proceedings Robotics and Automation, Arizona, pp 692-694, (1989).
[14] D. Cliff, I. Harvey, and P. Husbands, "Evolutionary robotics," IEE Colloquium on Design and Development of Autonomous Agents, London, UK, pp 1/1-1/3 (1995).

Evolving Electronic Circuits For Robotic Control

Evolving Electronic Circuits For Robotic Control

M. Beckerleg, J. Collins
Engineering Research Institute
Auckland University of Technology, Auckland, New Zealand
Email: mark.beckerleg@aut.ac.nz

***Abstract*- This paper details the implementation of a hardware genetic algorithm used to evolve a digital circuit capable of controlling the motion of a robot. The digital circuit is a virtual FPGA comprised of functional elements layered in four columns. The virtual FPGA is configured by a bit stream generated by the hardware genetic algorithm which supports only mutation. The robot is in the form of a simulated balancing beam which has 32 outputs connected to the digital circuit and one input for motor control. It was found that after an average of 40000 generations a digital circuit could be evolved to balance the beam for more than five minutes.**

I. Introduction

Evolving electronic circuits have been advanced from the generation of simple circuits through to more complex functional systems such as robotic navigation [1, 2]. In this paper we show how a digital controller circuit can be evolved that can be used to control a robotic system in the form of a simulated ball-beam. The simulation is modelled on a physical beam (Figure 1), which has 21 ball position sensors, a four pole stepper motor and a curved beam.

The complete system (Figure 2) is implemented on a commercially available FPGA and comprises three major subsystems: a NIOS processor which provides the robotic simulation, fitness evaluation, control of the evolutionary process and data transfer of the results to a PC; a virtual FPGA (VFPGA) which is an evolutionary capable digital circuit using a four layer Cartesian based architecture with a functional element at each node; and a hardware based genetic algorithm (HGA) with memory storage, random number generation and chromosome mutation capabilities.

Figure 1 Physical Beam

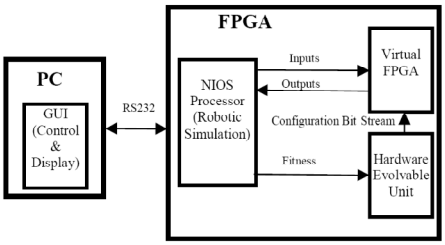


Figure 2 Complete System

The results of the simulation are fed into a PC to allow the recording of data and control of the experiment. The simulation models the behaviour of the ball and produces ball position, ball speed and beam position as inputs to the digital circuit. The digital circuit produces an output that controls the direction of the beam motor.

II. Background

Evolutionary computation mimics biological evolution in which populations evolve to best suit their environment. Initially the process generates a population of individuals (chromosomes) where the chromosome is a possible solution to a problem. Each individual is evaluated to determine how well it solves the assigned problem (fitness), the best individuals are kept (selection) and these individuals are used to create new offspring (reproduction). The process of finding fitness, selection and reproduction are repeated until the required fitness is reached.

Programmable logic devices such as a FPGA are programmed with a configuration bit stream which describes the hardware structure that is to be created inside the FPGA. In comparison with evolutionary computation, where the chromosome is a possible solution, the chromosome in evolvable hardware is a possible circuit described by the configuration bit stream. This bit stream can be modified with standard genetic algorithms, then downloaded into the FPGA and the ensuing circuit tested for fitness. The process is repeated until a suitable result is achieved, thus the hardware itself evolves.

The first application of evolvable hardware was performed by Thompson [3, 4] when he evolved a tone discriminator using evolutionary techniques on the Xilinx XC6216 FPGA.

The XC6216 is suited to evolution because it has a non-destructive architecture and has the ability to have partial reconfigurability. However it is no longer commercially available and subsequent FPGAs have complex routing systems that allow outputs to be connected to outputs, thus incorrect routing will quickly cause permanent damage. In a normal development environment destructive configurations would be avoided by the compiler, however the evolutionary process randomly mutates the configuration bit streams allowing destructive configurations to be produced.

Three different methods have been used to overcome this problem. The first is to use a genetic compiler [5] which filters out destructive or unreliable bit stream parameters generated by the evolutionary process. The second is to use genetic programming in which the actual program evolves rather than the configuration bit stream. Several techniques have been employed to improve the efficiency of this type of programming [6]. The third method is to create a virtual FPGA with an internal architecture that cannot connect outputs to outputs.

A FPGA contains two main components, routing and functional elements (FE), which can be programmed to replicate any digital circuit design. It is therefore possible to design a VFPGA circuit which mimics a FPGA but is suited to evolution and can be downloaded into a normal FPGA. This allows the FPGA to contain non-evolutionary as well as evolutionary capable circuits. The VFPGA function is modified by its own configuration bit stream which can either be loaded externally from a PC via an external pin or internally via an internal processor or hardware genetic algorithm.

The first attempt at creating a VFPGA was the recreation of the original Xilinx XC6200 series [7]. Following on from this, new VFPGA architectures were developed that were more suited to evolvable hardware. The requirements for a VFPGA architecture [8] are that it be non-destructive and have a reduced configuration bit stream which reduces the evolutionary search space. The bit stream can be reduced by limiting the routing capabilities and by implementing the FE as a functional block or higher level abstraction, rather than a low level lookup table.

Two examples of reducing routing are; the S-block architecture [9] in which the S-block operates as either a logic gate or a routing connection, thus no separate routing information is required in the configuration bit stream; and layered structures that have a reduced routing requirement based on a Cartesian architecture [10]. An example of increasing the abstraction is the functional block which replaces the lookup table inside the FE [11, 12]. The FE can also be expanded to incorporate behaviours using a subsumption architecture [13, 14]. An example of this is where the VFPGA architecture is created to match individual tasks such as input, output or control. All of these methods allow the configuration bit stream to be reduced as it is describing a functional or behavioural level rather than a basic lookup table level.

III. Mathematical model

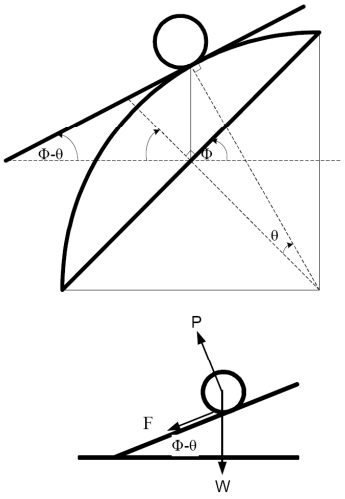


Figure 3 Beam Analysis

$$\ddot{\theta} = -\frac{mg}{R(m+\frac{I}{r^2})}\sin(\phi-\theta) \qquad (1)$$

Where
- m is the mass of the ball
- g is the gravitational acceleration
- R is the radius of curvature of the beam
- r is the radius of the ball
- I is the rotational inertia of the ball
- θ is the ball position (angle from the centre)
- φ is the beam position (angle from horizontal)
- F is the frictional force
- P is the reaction force
- W = weight force

The mathematical equation for the beam is shown in Figure 3 and Equation 1. Note the full mathematical analysis and modeling of the beam is described in a co-authored paper [15]. The beam position φ is related to the motor position. The motor is a four pole stepper motor with a 110 degrees/ second maximum rate of change. The simulator uses equation 1 to calculate the speed and position of the ball every 8 ms. This time period corresponds to the maximum rate at which the stepper motor can be pulsed.

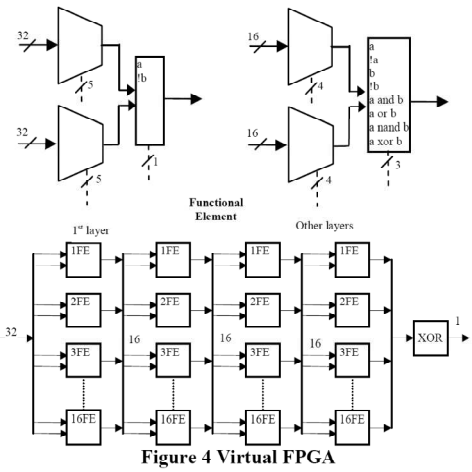


Figure 4 Virtual FPGA

IV. Virtual FPGA

The VFPA (Figure 4) has a four layer architecture with a grid pattern of sixteen FEs [16]. The first layer has 32 inputs and 16 outputs, the second to fourth layers have 16 inputs and 16 outputs, the fifth layer is an Exclusive OR which combines the sixteen outputs to provide the motor direction information. Each FE comprises of two multiplexers for routing two of possible 16/32 input signals and a functional block which provides functional logic. The configuration bit stream for each FE is eleven bits; the first layer uses 10 bits for the multiplexers and 1 bit for the functional block; the other layers use 8 and 3 bits respectively.

Each layer is configured independently of the others and requires a total of 178 bits (176 for the FE plus 2 for the hardware GA to adapt the mutation rate. The complete VFPGA will require 712 bits.

V. Hardware Genetic Algorithm

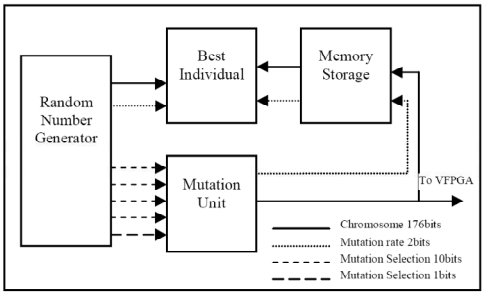


Figure 5 Evolvable Unit

The HGA comprises four evolutionary units, one for each layer of the VFPGA. A unit consists of a random number generator, memory storage for individuals and a mutation unit (Figure 5). The HGA is controlled by the NIOS processor.

On reset the 178bit random number generator creates four individuals which are separately loaded into the VFPGA. The VFPGA inputs and outputs are linked to the simulation allowing the individual to be evaluated for fitness. If the best fitness of the four offspring has the same or better fitness than the original then it is retained and the process is repeated.

The random number generator outputs the chromosome, the mutation rate and 4 10 bit numbers which are used to generate the mutation. The 10 bit number (range 0 to 1023) is used to select which bit in the chromosome will be mutated, which gives a possible mutation rate of 1.7%. The mutation rate is 2 bits thus the total mutation rate can range from 1.7% to 6.8%. The mutation rate itself is also mutated so that it will vary as well.

VI. VFPGA Configuration

The status of the ball and beam is connected to the inputs of the VFPGA and the VFPGA output is used to define the motor direction. The actual beam has a 4 pulse stepper motor capable of 110 degrees of movement per second and 21 ball position sensors showing the position of the ball. Originally the VFPGA was designed with 16 inputs, 15 used for the ball position and 1 for the ball speed, (left or right). On experimentation it was found that the VFPGA could not be evolved to balance the ball. An analysis of the behaviour revealed that not only ball position and speed, but also the beam position is an important component required for the VFPGA controller. These parameters were fed into an expanded VFPGA with 32 inputs which were divided into 19 ball positions, 3 ball speeds and 10 beam positions.

VII. Results

The system as previously shown in Figure 2 was run, with the generation and fitness logged every 1000 generations. The run was set to end when the beam had reached a fitness of 500 seconds with the fitness determined by the time taken before the ball hits an end stop. At this point the final chromosome was stored and the simulation repeated with a step by step recording of the beam and ball position and ball speed taken. This allowed the motion of the ball with relation to the beam to be monitored. Two motor speeds were evaluated, one set to pulse every four milliseconds, the other eight milliseconds (Figures 6 and 7). The faster motor required less generations to evolve a suitable fitness, however both performed satisfactory.

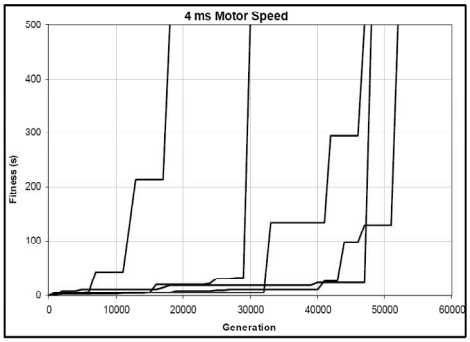


Figure 6 Best Fitness Motor Pulse 4 ms

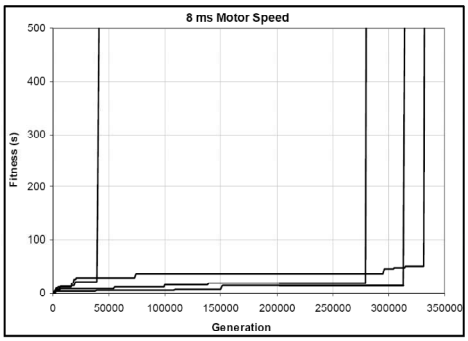


Figure 7 Best Fitness Motor Pulse 8 ms

A typical result would start with a fitness of 0 ms and then quickly find a fitness of 600ms. This is the time taken for the ball to roll down a stationary beam. The fitness would gradually improve in small steps until a fitness between 10 to 20 seconds was reached. At this point the fitness would increase in much larger increments, especially when the motor speed was set to 8 ms.

Several analyses were made of the ball and beam positions for a typical run of the successful individual. Initially the beam would either not respond or respond in the wrong direction for the ball motion. After several generations the behavior would change and the ball would roll forward, the beam would correct, then the ball would roll back into the end stop. This process would become more sensitive to the ball position as the evolution progressed, but eventually the ball would build up enough momentum to reach the end stop before the beam could correct it. A typical example of this is shown in Figure 8. This corrective process would evolve until a fitness of between 10 to 20 seconds was reached. However after reaching this value the fitness would then jump to a significantly higher value.

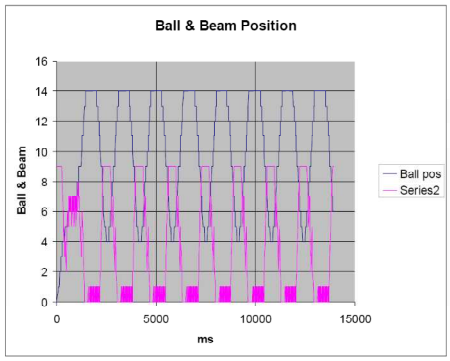


Figure 8 Relative ball beam positions

When these individuals where analysed it was found that the ball would spend most of its time towards one end of the beam jumping between a narrow range of positions, as the beam twitched between two set conditions. Eventually the ball would gain enough momentum to move away from this position and begin to travel towards the opposite end of the beam. The beam would then respond to bring the ball back to its semi stable state wher it would repeat the process. As this pattern could be repeated, rapid improvements in the fitness were achieved.

VIII. CONCLUSIONS

This paper has shown how an electronic circuit acting as a controller can be evolved to balance a simulated ball-beam system. A commercially available FPGA was used in the experiments to contain; an electronic circuit in the form of a virtual FPGA, a hardware genetic algorithm and a simulation running on a NIOS processor. The simulation provided ball and beam positions and ball speed to the VFPGA, whilst the VFPGA provided motor direction back to the simulation.

The VFPGA eventually evolved a circuit which had two closely spaced positions between which the ball oscillated. When the ball moved beyond these points the beam reacted in a self correcting manner to bring the ball back to its semi-stable state.

It was found that on average less than 40,000 generations were required to evolve such a circuit.

References

[1] K. C. Tan, C. M. Chew, K. K. Tan, L. F. Wang, and Y. J. Chen, "Autonomous robot navigation via intrinsic evolution," presented at Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on, 2002.

[2] K. C. W. TAN, L.F LEE T.H. AND VADAKKEPAT P., "Evolvable Hardware in Evolutionary Robotics," *Springer Science+Business Media B.V*, vol. 16, pp. 5-21, 2004.

[3] A. Thompson, "An evolved circuit, intrinsic in silicon, entwined with physics.," *Proc. 1st Int. Conf. on Evolvable Systems (ICES'96)*, pp. 390-405, 1997.

[4] A. Thompson, *On the Automatic Design of Robust Electronics Through Artificial Evolution*, 1478 ed, 1998.

[5] D. Levi and S. A. Guccione, "GeneticFPGA: evolving stable circuits on mainstream FPGA devices," presented at Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on, 1999.

[6] D. Montana, R. Popp, S. Iyer, and G. Vidaver, "EvolvaWare: genetic programming for optimal design of hardware-based algorithms," presented at Proceedings of Genetic Programming Conference (GP-98), 22-25 July 1998, Maidson, WI, USA, 1998.

[7] G. Hollingworth, S. Smith, and A. Tyrrell, "Safe intrinsic evolution of Virtex devices," presented at Evolvable Hardware, 2000. Proceedings. The Second NASA/DoD Workshop on, 2000.

[8] P. C. Haddow and G. Tufte, "An evolvable hardware FPGA for adaptive hardware," presented at Evolutionary Computation, 2000. Proceedings of the 2000 Congress on, 2000.

[9] P. C. Haddow and G. Tufte, "Bridging the genotype-phenotype mapping for digital FPGAs," presented at Evolvable Hardware, 2001. Proceedings. The Third NASA/DoD Workshop on, 2001.

[10] L. s. Sekanina, *Virtual Reconfigurable Circuits for Real-World Applications of Evolvable Hardware*, 2606 ed, 2003.

[11] T. G. W. Gordon and P. J. Bentley, "Towards development in evolvable hardware," presented at Evolvable Hardware, 2002. Proceedings. NASA/DoD Conference on, 2002.

[12] Z. Yang, S. L. Smith, and A. M. Tyrrell, "Digital circuit design using intrinsic evolvable hardware," presented at Evolvable Hardware, 2004. Proceedings. 2004 NASA/DoD Conference on, 2004.

[13] R. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of [legacy, pre - 1988]*, vol. 2, pp. 14-23, 1986.

[14] R. A. Brooks, "A robot that walks; emergent behaviors from a carefully evolved network," presented at Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on, 1989.

[15] J. Collins and M. Beckerleg, "An Evolved Controller for a Simulated Ball Balancing Beam " presented at 15th International Conference on Mechatronics and Machine Vision in Practice, Massey University, Auckland, New Zealand, 2008.

[16] J. P. Wang, C.H. Lee, C. H., "FPGA Implementation of Evolvable Characters Recognizer with Self-adaptive Mutation Rates," presented at International Conference on Adaptive and Natural Computing Algorithms ICANNGA'07, Warsaw, Poland, 2007.

Using a Hardware Simulation within a Genetic Algorithm to Evolve Robotic Controllers

Using a Hardware Simulation within a Genetic Algorithm to Evolve Robotic Controllers

M. Beckerleg, J. Collins

***Abstract*— This paper uses a novel method of implementing a genetic algorithm (GA) using a hardware simulation to evaluate the fitness of an individual for a robotic controller, rather than the normal practise of a software simulation. A simulation is required within a GA to model the actions of the robot and its environment in order to evaluate how well each individual within the population performs. Typically a simulation is written in software and executed sequentially on a processor. However, this paper implements the simulation as a digital circuit within a FPGA using a hardware description language (HDL). A comparison between identical hardware and software simulations is performed, resulting in the hardware simulation evolving a successful solution over seven hundred times faster than the software simulation. The robot is in the form of a balancing beam, the GA was implemented in hardware and the circuit driving the beam was a virtual FPGA.**

***Index Terms*—Evolvable Hardware, Genetic Algorithm, Hardware Simulation, Evolvable Robotics, Virtual FPGA**

I. INTRODUCTION

This paper uses the novel approach of using a hardware robotic simulation within a GA. The basis of a GA is to find a solution to a problem using evolution as a search engine. The GA uses natural selection to evolve a population of individuals where each individual represents a possible solution to a problem. The process is iterative and is comprised of three main sections: reproduction, fitness evaluation and selection. The selection process determines which individuals within the population will survive to the next generation based on their fitness. The reproduction process creates new offspring from the surviving parents using the genetic operators crossover and mutation. The fitness evaluation determines how well each individual within the population performs as a potential solution to the problem with this process being the most time intensive. In order to evaluate the fitness of an individual it must be tested either in real life or in simulation. As it is time consuming and potentially destructive to evolve a real life robot, a robotic simulation is used. Historically a simulation is run in software on a computer. If the simulation could be implemented in hardware on a FPGA, then the mathematical equations describing the simulation could be executed in parallel and there should be a decrease in the time taken for fitness evaluation.

Manuscript received June 03, 2011; revised August 12, 2011.

Mark Beckerleg is with the School of Engineering, AUT University, New Zealand. phone: +64-9-9219999, fax: +64-9-9219973 (e-mail mark.beckerleg@aut.ac.nz)

John Collins is with the School of Engineering, AUT University, New Zealand. (e-mail john.collins@aut.ac.nz)

This paper created two identical simulations for a robotic controller (Fig 1) with the first coded in software and the second implemented in hardware. The GA used to evolve the virtual FPGA which controlled the robot and the virtual FPGA itself were identical and were implemented in hardware. This enabled a valid comparison between hardware and software simulation to be performed.

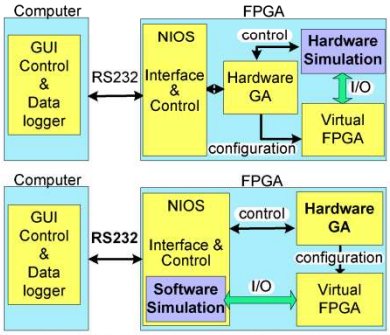


Fig 1. The two systems used to evaluate the software and hardware simulation.

The robotic platform used to test both simulations was a ball-beam system (Fig 2). This used a beam driven by a stepper motor to balance a ball between two end-stops. The beam had 19 sensors to determine ball position and a stepper motor which could move at 27.5 degrees per second to drive the beam. The beam itself was curved to make the system inherently unstable.

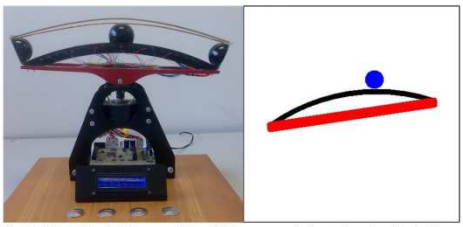


Fig 2. The physical beam with a GUI representation allowing the ball and beam to be dynamically observed during the evolution.

The main difficulty with creating a hardware simulation inside a FPGA is that unlike a computer, there is no arithmetic logic unit. All arithmetic formulae written in HDL will generate individual circuits to implement the arithmetic function. With floating point operations, a large number of the FPGA logic element resources are required for each calculation due to the complexity of dealing with

signed, mantissa and exponent parts. As there are typically many floating point calculations in a simulation, it becomes impractical to use this technique.

An alternative to floating point calculations is the use of integer arithmetic, which reduces the logic element resources required to implement the circuit within the FPGA. Trigonometric functions will also need to be implemented as an arithmetic approximation or a look up table, as they are difficult to implement in hardware.

The disadvantage of using integer calculations is the loss of precision compared to floating point calculations. In addition the algorithms must be checked to make sure that no arithmetic overflow occurs as the numbers are confined to 32 bits ($\pm$ 2 x 10^9). It is also important to ensure that the timing between the arithmetic calculations and the timing between the simulation and other systems is correct. Finally the hardware simulation must be integrated to the GA and the virtual FPGA.

To implement the simulation in hardware, the integer arithmetic calculations can be directly coded in Verilog HDL using the standard multiply and divide syntax.

II. Background

There has been a large amount of research in the use of GAs to evolve robotic controllers using both software and hardware GA's. However to the to the authors knowledge the use of a hardware simulation within these systems has not previously been used. Using evolution to create robotic controllers has been widely studied with advances in path planning [1, 2], obstacle avoidance [3, 4], tracking [5, 6] and even evolving the robot form itself [7, 8]. This paper advances the field by the use of a hardware robotic simulation to improve the completion time for the GA process.

A virtual FPGA was used to control the motion of the beam. This is a digital circuit which was evolved by modifying its configuration bit stream (CBS) which determined the circuit parameters within the virtual FPGA. This method, referred to as evolvable hardware, was first implemented by Thompson [9] when he evolved a tone discriminator on a evolutionary tolerant Xilinx FPGA. However this type of FPGA has been discontinued and it has become difficult to directly evolve a commercial FPGA by modifying its CBS.

The main requirements of an evolvable FPGA are a) scalability to enable large systems to be evolved, b) partial reconfigurability, where parts of the FPGA can be reconfigured while other parts are still running and c) non destructive architectures that are resilient to a random CBS. One solution that meets these requirements is the virtual FPGA with functional elements employed in a Cartesian based array that could be downloaded into the FPGA. The operation of the functional elements and their inputs are determined by the CBS, thus evolving this bit stream would change the operation of the virtual FPGA. Virtual FPGAs have been evolved for several applications including an adaptive equalizer with lossy data compression, [10] image processing [11] and character recognition [12].

The hardware GA used in the experiments was a mutation only GA (MOGA) implemented without the use of the crossover operator, which requires less FPGA resources allowing the complete system to be implemented in a relatively small Altera Cyclone EP1C12F324C8 device. This technique has been used previously in both hardware and software GAs with studies that showing that a MOGA can compare favorably against a normal GA [13]. An advantage of this technique is a reduction in chromosome damage caused by the crossover operator [14]. Various mutation only algorithms have been studied such as frame shift and translocation, once again finding a good comparison with a normal GA [15]. Several papers have used this method to evolve digital circuits.[12, 16, 17]. Other hardware GA systems have used crossover templates to minimize the number of two bit multiplexers for the crossover operation [18], while others have used pipelining and parallelism to develop a high speed hardware GA[19].

III. Mathematical Model

In the model of the beam (Fig 3), the beam position is measured as an angle φ from horizontal, and the ball position is measured as an angle θ from the centre of the beam. The full derivation for the mathematical model has previously been described by the authors [20]. The final equations for the ball acceleration are given in equations (1) and (2).

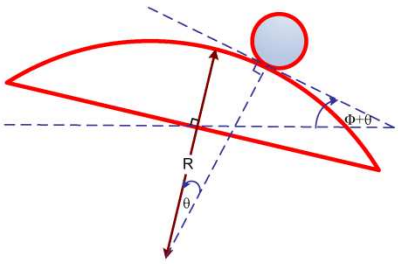


Fig 3. The ball and beam showing the relationships between the angles and motion.

$$\ddot{\theta} = A(\theta + \emptyset) \quad (1)$$

$$A = \frac{g}{R(1+\frac{I}{mr^2})} \quad (2)$$

From physical experimentation on the beam, the value for acceleration (a) of the ball was determined as a factor of the ball position (x) and beam position (b) in equation (3).

Placing this into the mechanical modeling we can determine the new position of the ball, dependant on its current position, velocity (v) and acceleration in equation (4), and the new speed of the ball dependant on its current speed and acceleration in equation (5). The simulation was set to a time period of 1 ms, in equations (6) and (7) and these were modified to give a divisor which was a multiple of two, enabling for efficiencies in the hardware implementation, in equations (8) and (9).

$$a = 12x + 2.8b \quad (3)$$

$$x_{new} = x + vt + \frac{at^2}{2} \quad (4)$$

$$v_{new} = v + at \quad (5)$$

$$x_{new} = x + \frac{v}{10^3} + \frac{12x+2.8b}{2x10^6} \quad (6)$$

$$v_{new} = v + \frac{12x+2.8b}{10^3} \quad (7)$$

$$x_{new} = x + \frac{1049v}{2^{20}} + \frac{101x+24b}{2^{24}} \quad (8)$$

$$v_{new} = v + \frac{786x+184b}{2^{16}} \quad (9)$$

Where

g - gravitational acceleration
I – moment of inertia of the ball
R - radius of curvature of the beam
m - mass of the ball
r - radius of the ball
θ - ball position (angle from the centre)
Ø - beam position (angle from horizontal)
x – ball position
v – ball velocity
b – beam position
a - acceleration of the ball

An investigation of the hardware simulation generated by the HDL compiler showed that no dividers and only four signed multipliers were used in the simulation circuit.

The beam state is defined by ball position, ball speed and beam position which were derived from the ball position sensors and the number and direction of the motor pulses. These parameter values were stored in a 32 bit word with 19 ball positions, 3 ball speeds and 10 beam positions, with only 1 bit active for each parameter at any one time.

Within the hardware simulation these parameters were encoded as integer values scaled up by 10000 to maintain accuracy. The first parameter was the beam position which was an angular parameter determined by the number of beam pulses. The beam had a total movement of $\pm 30^0$ with 270 pulse required to move the beam over the total range giving a movement per pulse of 0.22^0 and a pulse range from ±135. As the motor could only be pulsed every 8ms then the beam movement per 1ms was 0.0275^0.

The location of the ball was determined by 19 sensors. When the ball was midway between two sensors both became active. This enabled the resolution of the sensors to be doubled giving a range from ±19.

The ball velocity was estimated from the time between sensor signal changes, however the velocity estimation was quite poor thus only three ball velocities were used; moving left, moving right and almost stationary.

The complete system (Fig 4) shows the three blocks of the genetic algorithm; a) the hardware GA, b) the virtual FPGA and c) the hardware simulation, each with its associated control lines. These lines controlled the GA process, and allowed the transfer of fitness, best chromosome and ball states to the computer GUI. The virtual FPGA controlled the motion of the beam dependent on the ball-beam states, and it was this circuit that was evolved via a GA process on its CBS. The hardware simulation modeled the dynamics of the ball-beam so the fittest of the current virtual FPGA circuit could be evaluated. The hardware GA used a mutation only algorithm where the crossover operator was not used. This reduced the number of logic elements required for the hardware GA, enabling the complete system to be placed in a Cyclone device using less than 12000 logic elements.

IV. System Overview

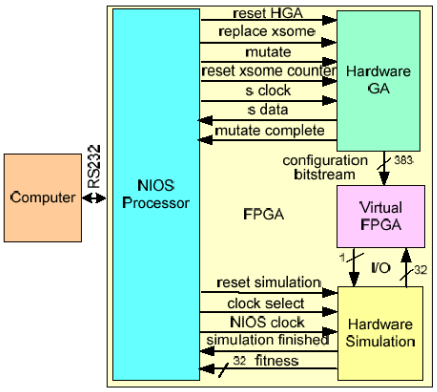


Fig 4. System overview showing connections between the NIOS processor and hardware subsystems

V. Hardware Genetic Algorithm

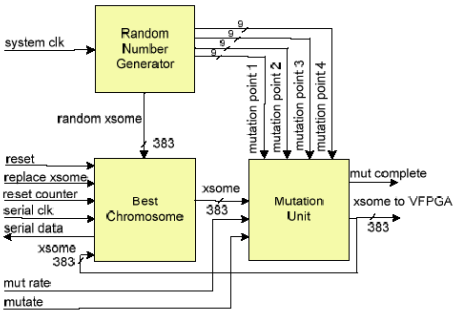


Fig 5. The hardware genetic algorithm showing the random number generator, chromosome storage, mutation unit and control lines.

The hardware genetic algorithm unit (Fig 5), had 3 blocks: random number generator, best chromosome storage and the mutation unit. The mutation random number generator used a linear feedback shift register to produce a 383 bit random number as well as four 9 bit random numbers. The best chromosome stored the current parent which could be sent to the mutation unit and sent to the computer via a serial link. The mutation unit could mutate 1 to 4 bits of the chromosome dependant on the selected mutation rate.

The operation was implemented as follows: on reset, a random CBS was generated, placed into the best chromosome memory, and then passed to the mutation unit. The mutation unit could mutate 1 to 4 bits within the CBS dependant on the fitness. The mutation point was set by a 9 bit random number giving a range of 0 to 511. As the CBS was only 383 bits, there was a possibility that a mutation would not occur (Table I). After mutation the CBS was sent to the virtual FPGA for evaluation. If the CBS had an equal or better fitness, it would be saved back in the best chromosome memory.

TABLE I
MUTATION RATE AND MUTATION PROBABILITY

Mutation Bits	Mutation Probability	Mutation Rate
1	75%	0 - 0.3%
2	94%	0 - 0.5%
3	98%	0 - 0.8%
4	99.6%	0 - 1.0%

VI. VIRTUAL FPGA

The circuit produced in a FPGA is determined by the CBS. Evolvable hardware uses a genetic algorithm to modify the CBS to produce a circuit that can be evolved. However a normal FPGA cannot be used, as a randomly generated CBS can destroy the FPGA. To overcome this problem, a virtual FPGA was created which was tolerant to random CBS. The virtual FPGA (Fig 6) used in this paper was based on the Cartesian based model consisting of functional elements (FE) which were grouped into five layers with the outputs of each layer feeding into the next.

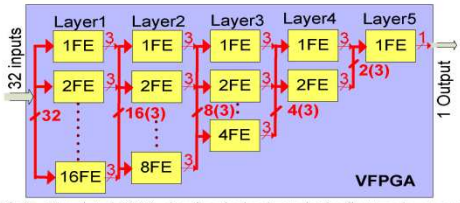


Fig 6. The virtual FPGA showing the hardware in the first two layers and the reducing layer structure within the device.

The configuration for the first and subsequent FEs is different (Fig 7). The first layer does not contain a function LUT. Each FE selects any 3 of the 32 inputs feeding them as a group of 3 bits to the next layer. The FE within the second and subsequent layers can select two groups from the previous layer and pass them to the function LUT which can perform Boolean and arithmetic functions on the two input groups. The operation of the multiplexers and the function LUT are controlled by the CBS which is 383 bits long.

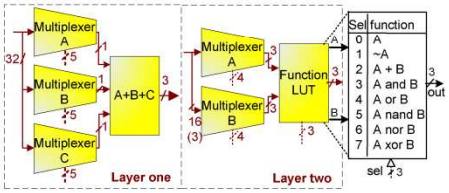


Fig 7. The FE in layer 1 multiplexes the inputs into groups of 3, while the FE in the second and subsequent layers select groups from the previous layer and performs Boolean and arithmetic operations on them.

VII. HARDWARE SIMULATION

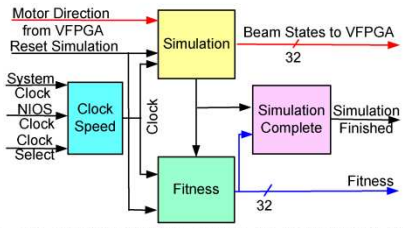


Fig 8. The hardware simulation showing the simulation and fitness calculation blocks with their associated control lines.

The hardware simulation (Fig 8) was comprised of four units: simulation, fitness, simulation complete and clock speed. The simulation unit contained the simulation's mathematical equations implemented in hardware, and the input-output to the virtual FPGA. The fitness unit had a 32 bit register which was incremented on every clock pulse, with each pulse equivalent to 1ms of simulation time. The clock speed unit switched the clock source between a 50MHz clock and a clock driven by the NIOS. Controlling the clock with the NIOS allowed the ball-beam states to be sent to the GUI, allowing the motion of the ball and beam to be displayed as well as enabling the simulation to be paused. The simulation complete unit was used to end the simulation when the fitness counter had reached 300 seconds.

When reset, the fitness counter was cleared and the simulation ball-beam parameters set to a starting position with the ball on the left side of the beam, with the beam set to an angle of 20^0. When not reset, the simulation mathematical equations were calculated on each clock cycle. The mathematical equations for the simulation were designed for a period of 1 ms, i.e. every clock pulse was equivalent to a one millisecond time period within the simulation.

After each clock pulse, the beam would be shifted left or right one motor step, dependant on the motor direction input from the virtual FPGA. The new integer ball speed, ball position and beam position would then be calculated, with these values then being converted into a thirty two bit binary format representing the new beam and ball state to be feed to the virtual FPGA. The simulation had an output control line to show when the simulation had finished. This was set whenever the ball position reached either of the two beam end-stops.

The fitness counter could be read by the NIOS processor at any time, with the value of the fitness counter being the time in milliseconds that the ball had remained balanced. The simulation finished line was also connected to the NIOS processor so that the fitness counter could be read at the end of a simulation

VIII. RESULTS

The first requirement was to ensure the software and hardware simulations operated in the same manner. To test this, a recording was taken of the ball position as it moved down the beam which was fixed at a 20^0 angle.

TABLE II THE MOTION OF THE BALL FALLING ON BOTH THE HARDWARE AND SOFTWARE SIMULATION.

Software Simulation				Hardware Simulation 5MHz				Time
Ball positn	ball speed	beam positn	Time (ms)	Ball positn	ball speed	beam positn	Time (ms)	Between Sensors
0	1	9	0	0	1	9	0	
0	2	9	184	0	2	9	184	
1	2	9	229	1	2	9	229	229
2	2	9	445	2	2	9	445	216
3	2	9	581	3	2	9	581	136
4	2	9	656	4	2	9	656	75
5	2	9	715	5	2	9	715	59
6	2	9	775	6	2	9	775	60
7	2	9	815	7	2	9	815	40
8	2	9	850	8	2	9	850	35
9	2	9	881	9	2	9	881	31
10	2	9	923	10	2	9	923	42
11	2	9	947	11	2	9	947	24
12	2	9	970	12	2	9	970	23
13	2	9	991	13	2	9	991	21
14	2	9	1015	14	2	9	1015	24
15	2	9	1033	15	2	9	1033	18
16	2	9	1050	16	2	9	1050	17
17	2	9	1070	17	2	9	1070	20
18	2	9	1085	18	2	9	1085	15
18	2	9	1096	18	2	9	1096	11

The table (Table II) shows the ball position starting on the left, with the ball speed initially stopped, then moving down the beam to the right with the beam in a fixed position. It can be seen from the table that both simulations are identical. On analysis, the ball slowly moved to the right, increasing in speed as the ball progressed along the beam. This is indicated by the decreasing time as the ball passed the sensors, starting off slowly and then increasing in speed due to the increased slope of the beam and the pull of gravity. Note the ball position sensors used for the GA were not evenly spaced and thus the time taken for the ball to pass between the sensors was not uniform either.

The second test was to compare the evolutionary progress of the two systems. The graphs of the fitness level relative to the current generation (Fig 9 and Fig 10) show that the evolution occurs in stages. A close investigation of the fitness and the movement of the ball showed that there were five stages to the evolutionary process. In all these stages it should be remembered that the beam has only two speeds, left and right. These stages are: Stage I balancing less than one second, the ball simply moved down the beam with no response to the motor. Stage II balancing less than two seconds, there was a jittering of the beam around a static position not dependent on the ball position. Stage III balancing less then ten seconds, there are several jitter points that are linked to the ball position. Stage IV balancing less than 300 seconds, the beam moved in such a fashion as to slow the ball down and kept it relative stationary for long periods of time. When the ball did move the beam would track it and bring it back to a relatively stationary period again. Stage V was a successful solution. The behaviour of the ball-beam was such that the ball would be semi stationary and move around a jitter point on the beam. Eventually the ball would break from this spot and move to the opposite side of the beam. The beam would then move to correct, and bring the ball back to the original location where the pattern would repeat indefinitely.

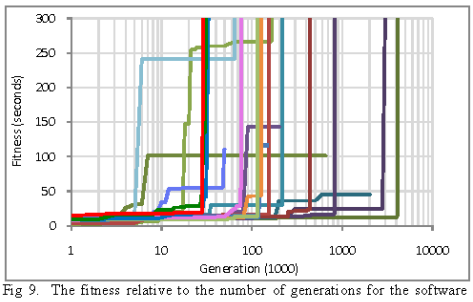


Fig 9. The fitness relative to the number of generations for the software simulation.

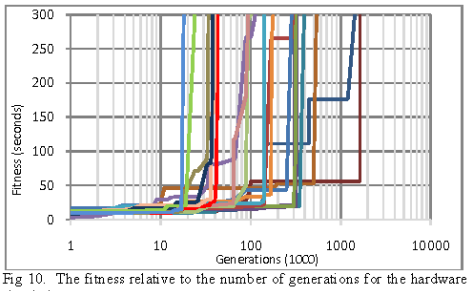


Fig 10. The fitness relative to the number of generations for the hardware simulation.

The final test was to compare the time taken for a successful evolution. These were plotted (Fig 11 and Fig 12), and the times compared. The average time for a successful evolution using a software simulation was 80,000 seconds, whereas the hardware simulation for this time was reduced to 110 seconds. Thus the hardware simulation could evolve a successful circuit on average 700 times faster than an identical software simulation.

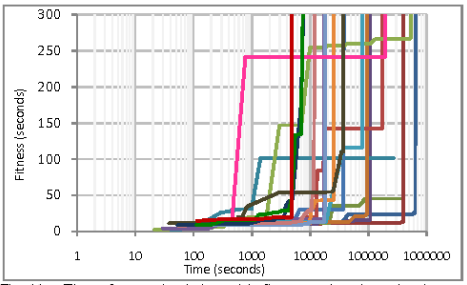


Fig 11. The software simulation with fitness and v time showing an average time of 50 thousand seconds to a successful evolution.

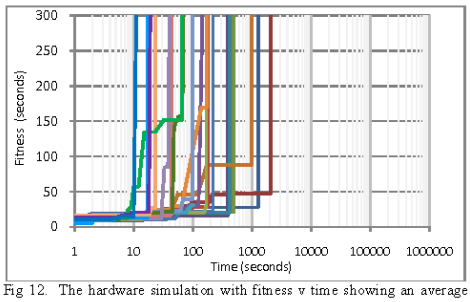


Fig 12. The hardware simulation with fitness v time showing an average time of eighty seconds to a successful evolution.

IX. Conclusion

A hardware simulation replicating a balancing beam has been successfully implemented. This simulation has been used in a hardware GA to evolve a virtual FPGA that was capable of balancing the ball on the beam for more than five minutes. A comparison between identical software and hardware simulations was performed with both systems behaving in an identical manner. It was found that the hardware simulation could evolve successful circuits over 700 times faster than the software simulation.

References

[1] D. A. Ashlock, T. W. Manikas, and K. Ashenayi, "Evolving A Diverse Collection of Robot Path Planning Problems," in *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, 2006, pp. 1837-1844.

[2] K. Daehee, H. Hashimoto, and F. Harashima, "Path generation for mobile robot navigation using genetic algorithm," in *Industrial Electronics, Control, and Instrumentation, 1995., Proceedings of the 1995 IEEE IECON 21st International Conference on*, 1995, pp. 167-172 vol.1.

[3] Y. Z. Renato A. Krohling, and Andy M. Tyrrell, "Evolving FPGA-based robot controllers using an evolutionary algorithm," 2002.

[4] A. M. Tyrrell, R. A. Krohling, and Y. Zhou, "Evolutionary algorithm for the promotion of evolvable hardware," *Computers and Digital Techniques, IEE Proceedings-*, vol. 151, pp. 267-275, 2004.

[5] K. C. Tan, C. M. Chew, K. K. Tan, L. F. Wang, and Y. J. Chen, "Autonomous robot navigation via intrinsic evolution," in *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, 2002, pp. 1272-1277.

[6] R. R. Cazangi, C. Feied, M. Gillam, J. Handler, M. Smith, and F. J. Von Zuben, "An evolutionary approach for autonomous robotic tracking of dynamic targets in healthcare environments," in *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, 2007, pp. 3654-3661.

[7] T. Koyasu and K. Ito, "Acquisition of the body image in evolution -Role of actuators in realizing intelligent behavior," in *Modelling, Identification and Control (ICMIC), The 2010 International Conference on*, pp. 859-864.

[8] M. Mazzapioda, A. Cangelosi, and S. Nolfi, "Evolving morphology and control: A distributed approach," in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, 2009, pp. 2217-2224.

[9] A. Thompson, "An evolved circuit, intrinsic in silicon, entwined with physics.," *Proc. 1st Int. Conf. on Evolvable Systems (ICES'96)*, pp. 390-405, 1997.

[10] M. Iwata, I. Kajitani, H. Yamada, H. Iba, and T. Higuchi, "A Pattern Recognition System Using Evolvable Hardware," *Lecture Notes In Computer Science*, vol. 1141, pp. 761-770, 1996.

[11] L. Sekanina, *Virtual Reconfigurable Circuits for Real-World Applications of Evolvable Hardware*, 2606 ed., 2003.

[12] J. P. Wang, C.H. Lee, C. H., "FPGA Implementation of Evolvable Characters Recognizer with Self-adaptive Mutation Rates," in *International Conference on Adaptive and Natural Computing Algorithms ICANNGA'07*, Warsaw, Poland, 2007, pp. 286-295.

[13] Z. Zhu, D. J. Mulvaney, and V. A. Chouliaras, "Hardware implementation of a novel genetic algorithm," *Neurocomput.*, vol. 71, pp. 95-106, 2007.

[14] T. L. Lau and E. P. K. Tsang, "Applying a mutation-based genetic algorithm to processor configuration problems," in *Tools with Artificial Intelligence, 1996., Proceedings Eighth IEEE International Conference on*, 1996, pp. 17-24.

[15] I. De Falco, A. Della Cioppa, and E. Tarantino, "Mutation-based genetic algorithm: performance evaluation," *Applied Soft Computing*, vol. 1, pp. 285-299, 2002.

[16] L. Sekanina and S. Friedl, "An Evolvable Combinational Unit for FPGAS," *Computing and Informatic*, vol. 23, pp. 461-486, 2004.

[17] L. Sekanina, T. Martinek, and Z. Gajda, "Extrinsic and Intrinsic Evolution of Multifunctional Combinational Modules," in *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, 2006, pp. 2771-2778.

[18] S. G. Shackleford B., Carter, R.J., Okushi E., Yasuda M., Seo K. Yasuura H. , "A High-Performance, Pipelined, FPGA-Based Genetic Algorithm Machine " *Genetic Programming and Evolvable Machines*, vol. 2, pp. 33-60, 2004.

[19] T. Maruyama, T. Funatsu, and T. Hoshino, *A Field-Programmable Gate-Array System for Evolutionary Computation*, 1482 ed., 1998.

[20] M. Beckerleg and J. Collins, "Evolving Electronic Circuits for Robotic Control.," in *15th International Conference on Mechatronics and Machine Vision in Practice* Auckland, New Zealand, 2008.

Evolving a Three Dimensional Lookup Table Controller for a Curved Ball and Beam System

Evolving a Three Dimensional Look Up Table Controller for a Curved Ball and Beam System

Mark. Beckerleg, John. Collins

Abstract—.**This paper presents a novel approach to the use of a genetic algorithm to evolve a three dimensional lookup table which acts as a robotic controller to balance a ball on a beam. The lookup table translates three ball-beam states, ball position, ball speed and beam position, into the motor speed and direction required to maintain the ball in balance. A population comprising these lookup tables was evolved by applying a genetic algorithm using tournament selection, two-point crossover and a mutation rate of two percent. Four different ranges of motor speeds within the lookup table were successfully evolved, each capable of maintaining the ball in balance for over five minutes.**

Index Terms—**Evolvable robotics, evolution of lookup table, lookup table based robotic controllers, ball and beam controller, genetic algorithms**

I. Introduction

This paper investigates the use of a genetic algorithm (GA) to evolve a ball and beam controller by evolving a population of lookup tables (LUT) used to control the beam. The system developed for this paper (Fig 1) consists of four parts: i) the graphical user interface (GUI), which displays the motion of the ball and beam with control and data logging capabilities, ii) the GA, which evolves a population of LUTs, iii) the simulation, which models the characteristics of the ball-beam system and iv) the LUT, which provides the new beam motor speed and direction depending on the current ball-beam state.

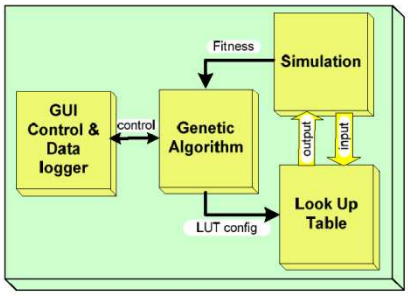


Fig 1. Block representation and connections between the four units that were implemented on a computer.

Manuscript received July 19, 2011; revised August 12, 2011.

Mark Beckerleg is with the School of Engineering, AUT University, New Zealand. phone: +64-9-9219999, fax: +64-9-9219973 (e-mail mark.beckerleg@aut.ac.nz)

John Collins is with the School of Engineering, AUT University, New Zealand. (e-mail john.collins@aut.ac.nz)

Typically a straight beam is used for a ball and beam apparatus, as it simplifies the control system algorithms that are required to balance the ball. However in this paper the beam is curved as this provides a more complex simulation model and algorithm and also means that the ball will never reach a static stable state with the motor stopped.

The simulation was modeled around a ball and beam system that was developed at AUT University for a student project (Fig 2). The physical beam was curved and had 19 infrared detectors that could determine the position of the ball, and a stepper motor that could alter the angle of the beam. The angular velocity of the beam was controlled by the number of pulses fed into the stepper motor per second. The maximum angular velocity was determined by the maximum pulse rate that the stepper motor could respond to. This was 125 pulses per second. The angular movement of 0.22 degrees per pulse gave a maximum angular velocity of the beam as 27.5 degrees per second.

Fig 2. The physical beam that the simulated ball and beam was modeled on.

This paper is organized as follows. In section two, background information about the ball and beam and the use of LUT in evolutionary computation is provided. Section three describes the mathematical model of the beam. In section four, the GA with its associated reproduction and selection schemes is explained. In section five, the simulation derivation and implementation is detailed. Section six contains the results of the evolution process and section seven contains the summary and conclusions.

II. Back Ground

Historically the ball and beam has been used as a standard laboratory apparatus to demonstrate control systems. It has also been used as a benchmark for research in control systems owing to its non-linear dynamics and behaviour. Several different control systems such as proportional integral differential (PID) control [1, 2], fuzzy

logic [3, 4], and neural networks [5, 6] have been studied using the ball and beam. The application of a GA to evolve ball and beam controllers has also been investigated. In particular a GA has been used in robotic controllers to evolve the rules and classes of a fuzzy logic controller [7, 8], the weightings and connectivity of artificial neural networks [9, 10], and the coefficients of a PID controller [11, 12]. However to the author's knowledge the use of a LUT for the beam controller evolved by a GA has not been investigated.

LUT's have been used in evolutionary computation in a variety of applications, although not as a robotic controller. Robotic simulations have been replaced by a LUT, reducing the amount of computation required when running the simulation and thus the evolution time [13, 14]. Researchers have evolved cellular automata by performing a GA on a LUT that held the cellular automata rules, to create two and three dimensional shapes [15]. Robotic controllers have been evolved with a LUT that was encoded with simulated DNA sequences in order to create robotic motion that drew motifs [16]. Research has also been performed on evolving the LUT found within a FPGA's functional elements using custom software to avoid destructive configurations [17].

The use of a GA to evolve a robotic controller based on a LUT has been performed by the authors on two robotic systems including a mobile inverted pendulum [18] and the gait of a hexapod robot [19].

III. Mathematical Model

In the model of the beam (Fig 3), the beam position is measured as an angle φ from horizontal, and the ball position is measured as an angle θ from the centre of the beam. The full derivation for the mathematical model has previously been described by the authors [20]. The final equations for the ball acceleration are given in equations (1) and (2).

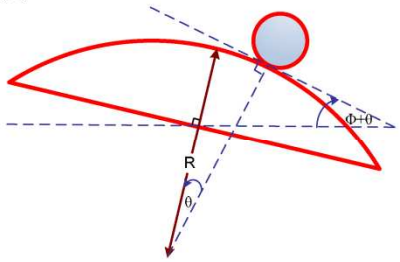


Fig 3. The ball and beam showing the relationships between the angles and motion.

$$\ddot{\theta} = A(\theta + \emptyset) \quad (1)$$

$$A = \frac{g}{R\left(1+\frac{I}{mr^2}\right)} \quad (2)$$

Where

g - gravitational acceleration
I – moment of inertia of the ball
R - radius of curvature of the beam
m - mass of the ball
r - radius of the ball
θ - ball position (angle from the centre)
Ø - beam position (angle from horizontal)
x – ball position
v – ball velocity
b – beam position
a - acceleration of the ball

From physical experimentation on the beam, the value for acceleration (a) of the ball was determined as a factor of the ball position (x) and beam position (b) in equation (3).

Placing this into the mechanical modeling we can determine the new position of the ball, depending on its current position, velocity (v) and acceleration in equation (4), and the new speed of the ball dependant on its current speed and acceleration in equation (5). The simulation was set to a time period of 1 ms in equations (6) and (7).

$$a = 12x + 2.8b \quad (3)$$

$$x_{new} = x + vt + \frac{at^2}{2} \quad (4)$$

$$v_{new} = v + at \quad (5)$$

$$x_{new} = x + \frac{v}{10^3} + \frac{12x+2.8b}{2x10^6} \quad (6)$$

$$v_{new} = v + \frac{12x+2.8b}{10^3} \quad (7)$$

IV. Genetic Algorithm

Evolutionary computation is an optimization process that autonomously searches through a sequence of possible solutions to a problem to find a solution that will adequately solve the problem. It is modelled on Darwinian evolution, 'survival of the fittest', where a population of solutions is evolved. Each solution is evaluated and given a fitness, and the solutions with a higher fitness are retained and used to create new solutions. These solutions are often referred to as individuals or chromosomes, and can be in many forms depending on the problem to be solved. A group of solutions is called a population. There are several forms of evolutionary computation, one of which is the GA.

The GA is a repetitive process with three parts including a) reproduction, where the genetic operators crossover and mutation are used to generate new individuals from the surviving population of individuals, b) fitness evaluation, which determines how well each individual within the population performs, and c) selection, which is the process that determines which individuals within the population (based on their fitness) will survive to the next generation.

A. Chromosome

The chromosome that was used for the ball and beam controller was a three dimensional LUT. The LUT's axial co-ordinates connected to the current ball-beam states of ball position (nineteen positions), beam position (ten positions) and ball speed (three positions). The parameter at each co-ordinate within the LUT was the desired motor speed and direction required to move the beam into a position that would maintain the balance of the ball (Fig 4).

The LUT was used to control the beam's motor depending on the beam states. This was achieved by connecting the simulation's current ball-beam states to the axis of the LUT. The parameter at that location was sent back to the simulation to control the simulation's motor speed and direction. The parameters within the LUT were

eleven discrete values ranging from 0 to 250, in steps of 25 giving a maximum of eleven motor speeds. If less speeds were required, these numbers were then broken into ranges, i.e. for two motor speeds, values less than 128 the motor was reversed, while numbers greater than 128 the motor was forward.

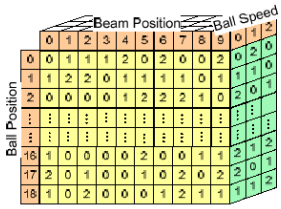


Fig 4. Three dimensional LUT showing 19 ball positions, 10 beam positions, 3 ball speeds.

The search space of a chromosome is the total number of combinations that the chromosome can have. The fitness landscape is the fitness level of each one of these chromosomes. In this study, the search space within the three dimensional LUT was dependant on the number of locations within the LUT, and the number of speeds that were employed at each location. The experiments were repeated with four ranges of motor speeds. These were two (left and right), three (left, stopped and right), five (two left, stopped and two right) and eleven speeds (five left, stopped and five right).

The total search space that the GA was required to search through was calculated using equation [8] and illustrated in Table I. It can be seen that the search space rapidly increased as the number of speeds increased. The exponent 570 was derived from the size of the LUT (19x10x3).

$$Search\ space = speeds^{\text{size of LUT}} = speeds^{570} \quad [8]$$

TABLE I SEARCH SPACE WITHIN THE LUT DEPENDENT ON THE NUMBER OF MOTOR SPEEDS.

Speeds	Search Space
2	3.9×10^{171}
3	9.1×10^{271}
5	2.6×10^{398}
11	3.9×10^{593}

B. *Reproduction and Selection*

The aim of the reproduction and selection scheme is to provide a high selection pressure, i.e. to move rapidly up the fitness landscape whilst maintaining population diversity. This is dependent on both the selection scheme and the reproduction method employed.

The reproduction method used two point crossover with a mutation rate of 2%. Two point crossover selects two random points within both parent's chromosomes and transposes the chromosome information at these points to create two new individuals. In this case the crossover points were selected only on the ball position axis (Fig 5).

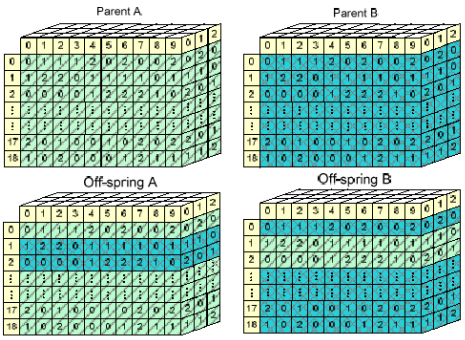


Fig 5. Two point crossover cut along the ball position axis.

There are many selection schemes that can be used within a GA, such as roulette, rank based, or tournament, with each method having its advantages and disadvantages. The selection process used in this GA was tournament, where the population was divided into groups. One individual within that group was selected for reproduction, depending on its fitness compared to the others within that group. The larger the group size, the higher the selection pressure; however with a larger group size diversity could be quickly lost. In this GA, a group size of two individuals was used.

C. *Fitness Criteria*

The fitness was determined by how long the ball remained balanced on the beam before hitting either end-stop. At the start of each test, the beam was placed in the horizontal position and the ball was at rest. The simulation was then run until either the ball hit an end-stop or 60 seconds had passed. Each individual was tested seven times with the ball positioned at seven different locations on the beam, giving a total maximum fitness of 420 seconds.

V. SIMULATION

The simulation used the equations as shown in equations [6] and [7]. The simulation was modeled on a 1ms time period. A new ball position and speed was calculated every 1ms time period. Correspondingly the beam movement was calculated over a similar period. The maximum beam movement was calculated from the real beam system, using two maximum motor speeds of 125 and 250 pulses per second, or a beam angular velocity of 22.7 and 45.4 degrees per second. The motor speed and direction was feed into the simulation which was used to calculate the new beam position. The new ball speed and position for the next 1ms was then calculated and fed back to the LUT. The 1ms time period of the simulation was used to give the real time that the ball was in motion.

VI. Graphical User Interface

The GUI (Fig 6) provided simple control of the GA and allowed the maximum fitness, average fitness, and duration of the simulation time to be recorded. The graphics of the beam could be physically turned on or off allowing the motion of the ball and beam to be observed. However when the graphics were enabled, the graphical drawing of the ball and beam on the display severely slowed down the computer program, therefore this feature was usually disabled.

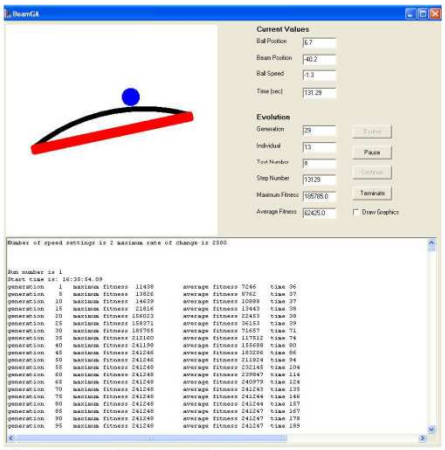


Fig 6. The graphical user interface

VII. Results

Initial experiments used a two dimensional LUT which used only the beam and ball positions. It was found that this information alone was not enough to provide a successful evolution. The LUT was modified to provide for a third parameter incorporating speed.

Two ranges of experiments were performed with two maximum stepper motor pulse rates. The first used 125 pulses per second which equated to a maximum beam angular velocity of 22.7 degrees per second. This is the speed of the actual beam motor and was at the limit at which the beam could control the ball. The second was 250 pulses per second which equated to a maximum beam angular velocity of 45.4 degrees per second. For each experiment, four ranges of motor speeds (two, three, five and eleven speeds) were evaluated.

The first experiments used eleven start positions lying between ±18 degrees from the top of the beam. The fitness level for these experiments never reached the maximum fitness. Under investigation it was found that the motor was not fast enough to prevent those balls starting at the extremes from hitting an end-stop. The experiment was changed to seven ball start positions lying between ±12 degrees from the top of the beam, with each individual tested seven times at each start position. A successful run occurred when the ball was balanced for 60 seconds; giving a maximum fitness of 420 seconds.

A. *Evolved motion of the ball.*

The graphs presented in figures 7 to 10 show the relationship between the fitness of the best individual within the population and the number of generations for the four ranges of motor speeds.

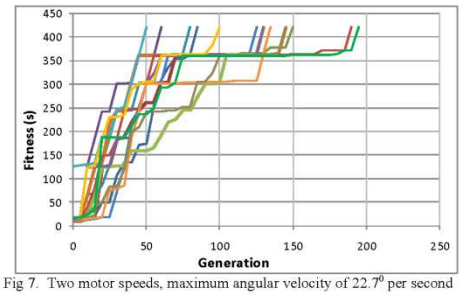


Fig 7. Two motor speeds, maximum angular velocity of 22.7^0 per second

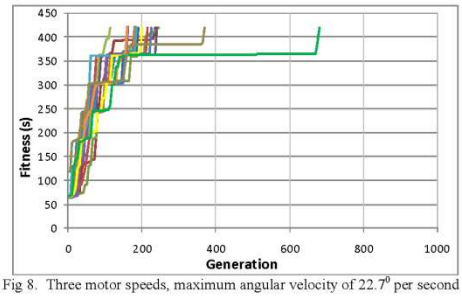


Fig 8. Three motor speeds, maximum angular velocity of 22.7^0 per second

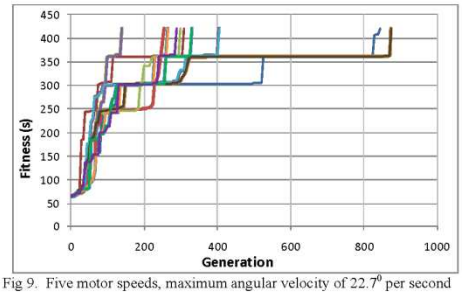


Fig 9. Five motor speeds, maximum angular velocity of 22.7^0 per second

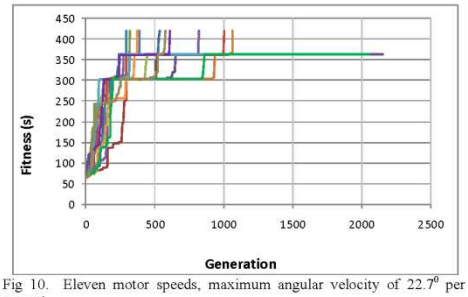


Fig 10. Eleven motor speeds, maximum angular velocity of 22.7^0 per second

The graphs show a step function in the fitness level as the evolution had to independently evolve seven start positions. It can be seen that the experiments with two motor speeds evolved to successful solutions in less generations and time than the other speeds.

The motion of the ball and beam was observed in the various stages of the evolutionary process using the graphical display. In the first stage, the ball would roll towards the beam end-stops with little reaction from the beam itself. In the next stage the beam would react to the ball movement, reversing the motion of the ball; however the ball would then roll to the opposite end-stop. During the following stage in the process, the beam moved in an oscillating pattern, causing the ball to stay balanced in between two points. However after five to ten seconds the ball would break free and gather too much speed for the beam to prevent the ball from hitting an end stop. In the final stage of evolution, the beam was able to keep the ball trapped between two points for the full 60 seconds.

This characteristic oscillation of the beam was seen in all motor speed ranges. With only two speeds, the beam moved in rapid oscillations to keep the ball steady. However with a larger number of speeds, the beam would move at a slower pace. Eventually the beam evolved to keep the ball motionless, with the use of an oscillation pattern for all seven start positions.

It can be seen from the graphs that there are two main plateaus in the fitness level near 320 and 360 seconds. These plateaus can be explained by the two start positions at the furthest point from the center of the beam. These are the most difficult points bring the ball to a stable oscillating condition, as the ball tends to gather a high speed and is difficult to capture. This plateau was more noticeable in the five and eleven motor speeds.

For the five and eleven motor speed range, the ball would not be balanced in the middle of the beam. Instead it would be gently moved to either end of the beam, and kept centered around that point. This trait can be explained by the placement of the ball sensors on the beam. When the beam was designed, the ball sensors were unevenly spaced with the sensors placed closer together at the ends of the beam and further apart in the middle of the beam. This was because it was thought that determining the ball's position and speed was more critical near the beam ends. Unintentionally however, this gave the evolved controller the best location of the ball and its speed near either end of the beam. Subsequently the evolved controller used the end locations to balance the ball. This characteristic was not seen with the two and three motor speeds experiments.

Because a simulation was used, when a test was started with the ball motionless in the center of the upright beam, the evolved solution kept the motor off, so the ball stayed perfectly balanced for the duration of the test. In the case of the two speeds as the motor could not be stopped, it would move the ball to a stable position.

B. *Evolved chromosome*

An investigation of successfully evolved chromosomes and the corresponding sequence of beam and ball motions showed different patterns for each evolved chromosome. This is because there were multiple ways of successfully balancing a ball. A successful evolution did not use a large part of the parameters in the LUT, especially at the extreme values of beam and ball positions. The ball simply tracked to a position on the beam, and beam oscillations around that point kept it in place.

A comparison of the maximum and average fitness (Fig 11) shows the maximum fitness increased in steps with the average fitness converging when the maximum fitness reached a plateau. At each plateau it was thought that as all the population had the same fitness, the population diversity had been lost. However an investigation of each chromosome revealed that this was not the case. This was backed by observation of the beam and ball motion at the plateau points. The evolution produced multiple solutions, although no individual chromosome had found a solution that would balance the ball when started in either, or both its first and last start position. Eventually this solution was found and the evolution was completed.

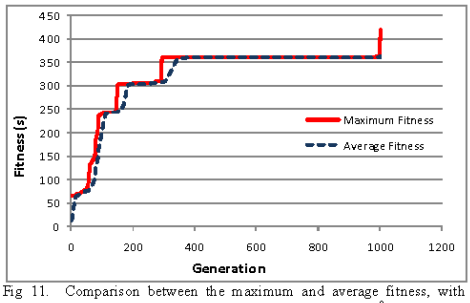


Fig 11. Comparison between the maximum and average fitness, with eleven motor speeds and a maximum angular velocity of 22.7°/sec

C. *Comparison of two maximum motor speeds*

Several hundred experiments were performed on both maximum motor pulse rates and speed ranges. Table II provides a comparison of these results showing the average fitness, number of generations and time the evolution was in progress at the end of the evolution. From this table it can be seen that the faster motor and minimum number of motor speeds had the best results in terms of the number of generations and the time taken to come to a successful evolution. It was noted that the time taken for the five and eleven motor speeds to successfully evolve was also acceptable despite the much larger search space. This was due to the constrained motion of the beam and the path that the ball took, with only a limited part of the chromosome being used for the beam control.

TABLE II COMPARISON OF THE AVERAGE FITNESS, AVERAGE NUMBER OF GENERATIONS AND THE AVERAGE TIME TAKEN TO EVOLVE

22.7 degrees/second			45.4 degrees/second		
Generation	Av fitness	Time (s)	Generation	Av fitness	Time (s)
118	347726	197	42	268456	35
268	364240	592	56	327891	76
398	357240	3624	98	351811	297
861	359427	25794	103	349563	467

A comparison of the four motor speeds within each maximum motor pulse rate is shown in figures 11 and 12.

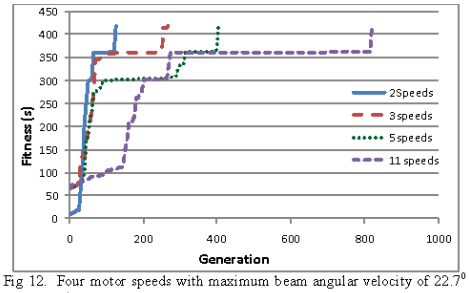


Fig 12. Four motor speeds with maximum beam angular velocity of 22.7^0 per second

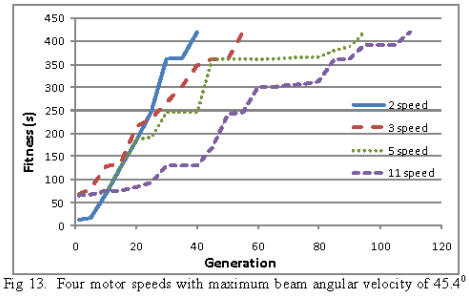


Fig 13. Four motor speeds with maximum beam angular velocity of 45.4^0 per second

From these graphs it can be seen that doubling the motor pulse rate had a significant improvement on the ability of the system to evolve, especially at the five and eleven speed range. The fitness plateau at 320 and 360 seconds can clearly be seen. All the solutions had difficulty with either one or both of the extreme starting points.

VIII. Conclusion

It has been demonstrated that a robotic controller for a ball and beam system based on a three dimensional lookup table can be evolved. While both motor pulse rates and all motor speed ranges were capable of being evolved to keep a ball balanced more than five minutes, the best evolutionary performance was achieved using a limited number of motor speeds and a higher motor pulse rate.

References

[1] C. Ka and L. Nan, "A ball balancing demonstration of optimal and disturbance-accomodating control," *Control Systems Magazine, IEEE*, vol. 7, pp. 54-57, 1987.

[2] F. G.-E. F.Gordillo, R.Ortega, and J.Aracil, "On the ball and beam problem: regulation with guaranteed transient performance and tracking periodic orbits," presented at the Proc of the International Symposium on Mathematical Theory of Networks and Systems, University of Notre Dame, IN, USA, 2002.

[3] E. P. Dadios, R. Baylon, R. De Guzman, A. Florentino, R. M. Lee, and Z. Zulueta, "Vision guided ball-beam balancing system using fuzzy logic," in *Industrial Electronics Society, 2000. IECON 2000. 26th Annual Conference of the IEEE*, 2000, pp. 1973-1978 vol.3.

[4] J. Iqbal, M. A. Khan, S. Tarar, M. Khan, and Z. Sabahat, "Implementing ball balancing beam using digital image processing and fuzzy logic," in *Electrical and Computer Engineering, 2005. Canadian Conference on*, 2005, pp. 2241-2244.

[5] K. C. Ng and M. M. Trivedi, "Neural integrated fuzzy controller (NiF-T) and real-time implementation of a ball balancing beam (BBB)," in *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, 1996, pp. 1590-1595 vol.2.

[6] P. H. Eaton, D. V. Prokhorov, and D. C. Wunsch, II, "Neurocontroller alternatives for ;fuzzy; ball-and-beam systems with nonuniform nonlinear friction," *Neural Networks, IEEE Transactions on*, vol. 11, pp. 423-435, 2000.

[7] A. G. B. Tettamanzi, "An evolutionary algorithm for fuzzy controller synthesis and optimization," in *Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century., IEEE International Conference on*, 1995, pp. 4021-4026 vol.5.

[8] K. Sung Hoe, P. Chongkug, and F. Harashima, "A self-organized fuzzy controller for wheeled mobile robot using an evolutionary algorithm," *Industrial Electronics, IEEE Transactions on*, vol. 48, pp. 467-474, 2001.

[9] R. Bianco and S. Nolfi, *Evolving the Neural Controller for a Robotic Arm Able to Grasp Objects on the Basis of Tactile Sensors*, 2829 ed., 2003.

[10] K.-J. Kim and S.-B. Cho, "Dynamic selection of evolved neural controllers for higher behaviors of mobile robot," in *Computational Intelligence in Robotics and Automation, 2001. Proceedings 2001 IEEE International Symposium on*, 2001, pp. 467-472.

[11] Z. Yi and Y. Xiuxia, "Design for beam-balanced system controller based on chaos genetic algorithm," in *Information Acquisition, 2004. Proceedings. International Conference on*, 2004, pp. 448-451.

[12] G. K. M. Pedersen and M. V. Butz, "Evolving robust controller parameters using covariance matrix adaptation," presented at the Proceedings of the 12th annual conference on Genetic and evolutionary computation, Portland, Oregon, USA, 2010.

[13] H. H. Lund and J. Hallam, "Evolving sufficient robot controllers," in *Evolutionary Computation, 1997., IEEE International Conference on*, 1997, pp. 495-499.

[14] H. H. Lund, "Co-evolving Control and Morphology with LEGO Robots.," in *Proceedings of Workshop on Morpho-functional Machines*, 2001.

[15] A. Chavoya and Y. Duthen, "Using a genetic algorithm to evolve cellular automata for 2D/3D computational development," presented at the Proceedings of the 8th annual conference on Genetic and evolutionary computation, Seattle, Washington, USA, 2006.

[16] G. Greenfield, "Evolved Look-Up Tables for Simulated DNA Controlled Robots," presented at the Proceedings of the 7th International Conference on Simulated Evolution and Learning, Melbourne, Australia, 2008.

[17] Y. Z. Renato A. Krohling, and Andy M. Tyrrell, "Evolving FPGA-based robot controllers using an evolutionary algorithm," 2002.

[18] M. Beckerleg and J. Collins, "An Analysis of the Chromosome Generated by a Genetic Algorithm Used to Create a Controller for a Mobile Inverted Pendulum," *Studies in Computational Intelligence*, vol. 76, 2007.

[19] J. Currie, M. Beckerleg, and J. Collins, "Software Evolution Of A Hexapod Robot Walking Gait," in *Mechatronics and Machine Vision in Practice, 2008. M2VIP 2008. 15th International Conference on*, 2008, pp. 305-310.

[20] M. Beckerleg and J. Collins, "Evolving Electronic Circuits for Robotic Control.," presented at the 15th International Conference on Mechatronics and Machine Vision in Practice, Auckland, New Zealand, 2008.

Software Evolution of a Hexapod Robot Walking Gait

15th International conference on Mechatronics and Machine Vision in Practice (M2VIP08),2-4 Dec 2008,Auckland,New-Zealand

Software Evolution Of A Hexapod Robot Walking Gait

J. Currie, M. Beckerleg, J. Collins
Engineering Research Institute
AUT University
Email: joncur96@aut.ac.nz

***Abstract* – This paper describes the process of evolving a Hexapod Robot walking gait within a simulated software environment. Initially a 3D mathematical model of the robot was created using Matlab, simulating full motion of each the robot's six legs. Each leg has three Degrees Of Freedom (DOF), allowing the robot to move in both lateral and rotational directions. The simulation allows the robot's movements to be determined, based on a repeated sequence of static leg positions, and was used with a Genetic Algorithm (GA) to evolve walking gaits for the robot. The walking gait is described by a chromosome which is an 18x9 Look Up Table (LUT) that lists the angular position of all eighteen servo motors over a sequence of nine discrete static positions, which will describe the walking gait. Each position in the LUT has twenty discrete states ranging from −45° to +45°, allowing a flexible range of achievable motions, while maintaining sensible evaluation limits. Successful evolution of these gaits was performed within 700 generations.**

I. INTRODUCTION

The aim of this research is to evolve a walking gait for a simulated Hexapod Robot, such that it can walk forward in a straight line. Previous research has been conducted in this area for typically rectangular hexapod robots [1-3], whereas this research will present evolution of a walking gait for a circular hexapod robot, with legs spaced 60° apart.

Designing the walking gait of a multi-legged robot is a complex multi-dimensional control problem [4, 5], with factors including centre of gravity, and surface friction to be accounted for. Designing a gait for a hexapod robot requires the coordination and simultaneous movement of all six legs, within a cycle of leg activations [6].

The manual configuration of this walking gait is a time consuming process, requiring specialist knowledge of the robot architecture, and can often result in sub-optimal performance. An example is the Sony AIBO robot, in which evolved or learned gaits consistently outperformed hand-tuned gaits, in terms of gait speed. The robots were used within Robot Soccer tournaments, and thus speed was one of the deciding factors of the victorious team [7].

Within this research, a Genetic Algorithm (GA) has been chosen to evolve the walking gait, based on the suitability of a GA for developing locomotion mechanisms [8]. This is a reasonable assumption based on Evolution Theory [9], which it stated that locomotion mechanisms of many life forms resulted from the process of natural evolution.

A GA follows the principles of natural evolution [9, 10]. Initially a random population is created, with each individual known as a "chromosome", and representing a possible walking gait of the robot. The population is then expanded by generating "offspring", chromosomes which are created from the variables within two parents' LUT (genes). Genetic operators perform this procreation process, by combining the genes of each parent into an offspring's LUT. A selection process known as "survival of the fittest" now places the offspring in direct competition with their parents, by placing the chromosomes into the mathematical model simulation, and returning a "fitness value". The best chromosomes are kept, and the worst discarded, and the process continues iteratively until a suitable table of solutions is found [11].

The robot has been modelled within Matlab, in order to reduce the evaluation period of each GA generated solution. The simulated robot is based on a physical robot constructed from a Lynxmotion kitset shown in Fig.1, with the electronics purpose built for this research. The robot has six legs, spaced evenly around a circular body. Each leg has three DOF, represented as a pelvic joint, a hip joint, and a knee joint.

The objective of this paper is to describe the mathematical model created to simulate the physical robot, and detail the GA used to evolve walking gaits.

Figure 1. Lynxmotion Hexapod Robot BH3-R

15th International conference on Mechatronics and Machine Vision in Practice (M2VIP08),2-4 Dec 2008,Auckland,New-Zealand

II. MATHEMATICAL MODEL

A. *Static Mathematical Model*

This is the model built to represent the physical position and orientation of the robot based on the angles of all eighteen servo motors. Static refers to modelling the robot in only one position, with no movement taken into account.

Initially, there are two assumptions this model relies on:

1) The robot cannot balance on two legs, and therefore there must be always at least three legs on the ground.
2) The centre of the mass of the robot is always at the origin of the robot's body.

Each leg is divided into four coordinate points, as shown in Fig. 2, and these are defined by the geometry of the robot, and the angles of the associated servo motors. The calculations to solve the coordinate points take the following form:

$$X(Pelvis) = OriginX + Radius \times cos(\delta) \quad (1)$$

$$X(Hip) = X(Pelvis) + Pelvis \times cos(\gamma + \delta) \quad (2)$$

where OriginX is the X coordinate of the centre of the robot body, Radius is the physical radius of the robot's centre body, while δ describes the angle where each leg connects to the robot body, relative to the front of the robot. Pelvis is the physical length of the pelvis of the robot and γ is the angle of the pelvis servo motor.

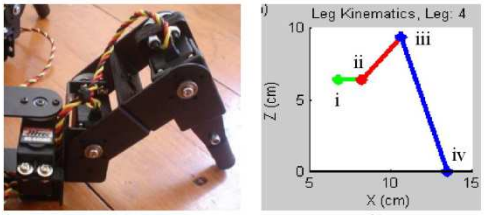
The equations for the knee and foot follow the same conventions, including the angles of their respective servo motors and body parts.

This initial part of the model solves for the coordinates of the robot as if it the body were sitting on the ground, but the feet are allowed to be at any height (including under the ground). This allows the robot's tilt (φ) and direction of tilt (θ) to be evaluated, based on the three feet that would be supporting the robot. To find the three supporting feet, the coordinates of the three lowest feet are checked to see whether they contain the centre of mass of the robot. Based on assumption (2), it would therefore mean the triangle containing the three coordinate points of each foot, would also contain the centre of the robot. In order to determine this, a "Point In Triangle" test was required. This is done by computing the barycentric [12] coordinates (u,v) of the triangle, and testing for the following:

$$u \geq 0, v \geq 0, u + v \leq 1 \quad (3)$$

When Equation (3) is valid, the centre of the mass is found to be within this triangle. If the lowest three feet do not contain the centre of mass (for example where the lowest three feet are on the same side of the robot), the calculation is iterated using substitute feet until a valid solution is found.

Once the triangle containing the three supporting feet had been established, the equation of plane containing this triangle was found by solving for the normal to this plane.



(a) (b)

Figure 2. (a) Actual leg of robot, (b) Model leg of robot showing the Pelvis Servo (i), Pelvis stick (green line), Hip Servo (ii), Femur (red line), Knee Servo (iii), Tibia (blue line) and Foot (iv).

Solving a Spherical to Cartesian transformation on the normal vector resulted in the φ and θ of the robot body. The next step was to rotate and lift the robot so the plane connecting the supporting feet was flat, at Z = 0. This was achieved by defining a new axis system, defined as three unit vectors:

$$N = \frac{normal\ vector}{Rho} \quad (4)$$

$$R = \frac{(0,0,1)\ cross\ normal\ vector}{cos(Phi)} \quad (5)$$

$$S = N\ cross\ R \quad (6)$$

where N is a unit vector pointing in the direction of the normal, R is a unit vector orthogonal to the normal, and orthogonal to the direction the plane must be rotated to be flat, and S is a unit vector orthogonal to both N and R. This new axis system is individual to the tilt and direction of the robot, and allows the existing coordinates to be translated into a normal Cartesian system, using the following:

$$q = Any\ (X,Y,Z) of\ the\ robot \quad (7)$$

$$r = q\ dot\ R \quad (8)$$

$$s = q\ dot\ S \quad (9)$$

$$t = q\ dot\ N \quad (10)$$

$$Translated\ X = (r \times R) + (s \times -cos(\theta)) \quad (11)$$

Equations 8-10 take the magnitude of the (X,Y,Z) components in relation to the new axis system, while Equation 11 translates the X coordinate to a new position, as if the robot is now standing on the ground. Similar equations translate the Y and Z components. Note that the translation is not performed on the X and Y components if φ = 0° (the robot is flat), and instead only on the Z component.

At this stage the robot is now standing on the ground, with the three supporting feet at Z = 0, with all coordinate points reflecting the tilt and direction of tilt of the robot body. It was

found that due to tilting the robot that one or more of the feet not supporting the robot could end up with $Z < 0$. In order to compensate for this, the model iterates until all feet are at $Z \geq 0$. The resulting model is shown in Fig. 3.

B. *Dynamic Mathematical Model*

To simulate the robot fully, a dynamic model was built around the static model, which would take consecutive static positions of the robot's legs, and make a prediction on the resulting movement. Movement was characterised by the following measurable values:

i. Lateral movement, with respect to X and Y.
ii. Heading, the direction the robot is facing.
iii. Height, the height of the centre of the robot.

An important assumption was also made:

3) A minimum of two legs must move if the body of the robot is to move (laterally or heading).

Assumption (3) is made to take into account the forces of friction on the feet. This means if one foot moves while the others do not, it is assumed this foot will slip across the ground, rather than move the robot. In contrast, it is expected that if two or more feet move, then the body of the robot will move, and the feet will stay approximately static.

The model works by assessing two consecutive static positions of the robot. To increase the accuracy of the model, the two consecutive positions are modelled in the same way the controller software on the physical robot can modify the servo angles, by the maximum and minimum change of 4.5°.

An incremental routine is setup on the robot controller, allowing the angle change of 4.5° per increment, per servo, iterating until all servo motors have reached their final desired positions. This same incremental routine is thus written into the dynamic model.

The movement of the robot is calculated by each increment, which can extend to over one hundred and fifty increments for a nine static position LUT. Movement is calculated the following process:

1) The first static position is programmed so that all servo motors are at 0°, such that the robot stands like in Fig. 1, facing forward.
2) For each subsequent static position, defined by the increment routine, the associated servo angles are placed into the static model to determine the coordinates of all parts of each leg and the feet supporting the robot. At this point the robot height can also be calculated from Equation (4) and:

$$Height = BaseHeight + |(X,Y,Z)\ dot\ N| \qquad (12)$$

where (X,Y,Z) are the coordinate points of any supporting foot and BaseHeight = 5.1cm, and this is also the height of the centre structure of the robot body, as depicted in Fig. 1.

3) The next step is to calculate the lateral movement of the robot. If there are feet which have supported the robot over the two consecutive static positions, and the feet have moved, then the robot may have moved as well. To calculate this, the supporting feet displacement vectors are added, to result in a vector which describes the net movement of the robot.
4) Next, the change in heading of the robot (dØ) is performed, by calculating the angle between two vectors:

$$VecA1 = tan^{-1}\left[\frac{Position\ 1\ Foot(Y) - Centre(Y)}{Position\ 1\ Foot(X) - Centre(X)}\right] \qquad (13)$$

$$VecA2 = tan^{-1}\left[\frac{Position\ 2\ Foot(Y) - Centre(Y)}{Position\ 2\ Foot(X) - Centre(X)}\right] \qquad (14)$$

$$d\emptyset = VecA1 - VecA2 \qquad (15)$$

where Position [1,2] describe the respective static positions, Foot(X or Y) describes the coordinate of a supporting foot, and Centre(X or Y) describes the coordinate of the centre of the robot.

5) The resulting change in heading of the robot is then defined as the average of dØ, caused by the movement of each supporting foot. The average was chosen to accommodate the random movements associated with a GA created LUT.
6) At this point the robot's height, lateral movement and rotation have been calculated, but the supporting feet have also moved, which may not be the case. For example, if the robot body was to move, based on the displacement vectors of the supporting feet, then the feet must have remained stationary in order for the body to have moved (based on friction). In order to calculate back where the feet should be based on the new heading and position of the robot body, the new positional data is fed back into the static model, and all four leg coordinates for each leg recalculated.

This sequence is repeated from steps (2) to (6) for each increment. A typical result is shown in Fig. 4.

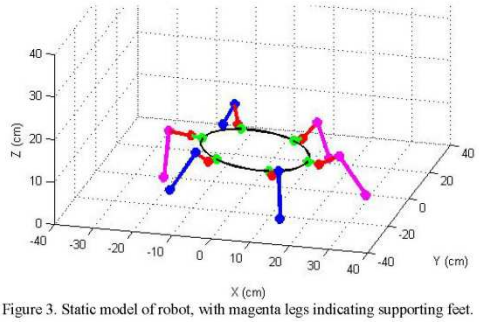


Figure 3. Static model of robot, with magenta legs indicating supporting feet.

III. GENETIC ALGORITHM

A. *Fitness Evaluation*

In order to analyse the performance of a particular LUT with regards to movement, a single output parameter is required from the simulation, as a measure of performance for the GA to use. The ultimate goal of this research is for the robot to walk forward in a straight line, and thus the performance criteria are centred on this goal.

Several different fitness calculation methods were tried, and these are detailed in the Results section. The final method chosen is based on two weighted requirements, the robot moving forward one metre in a straight line, after a maximum of one hundred iterations of the optimised LUT, and the stability and efficiency of the evolved gait.

The target location is directly in front of the robot and the distance is chosen based on direct experimentation with the real robot and walking gaits, but as yet unattained with hand tuning of the gaits. This will account for 75% of the final fitness score.

Stability is measured by using the standard deviation of the body height of the robot, across the simulation of the evolved gait. Efficiency is the number of steps the robot took to walk during the gait. Gaits which move left and right before moving forward are deemed inefficient, and thus will score a lower fitness. Both these factors are weighted appropriately to make up 25% of the final fitness.

The procedure of calculating the fitness is as follows:

1) From the dynamic model, the distance the robot moved, the direction of movement, and heading of the robot is passed to a fitness evaluation function.
2) The position of the robot is then calculated for every iteration of the LUT, starting at a single iteration (n = 0) and working towards 100 iterations (n = 99). For each iteration, the x coordinate of the robot is checked to see if it has remained between ±30cm, the vertical boundaries, thus limiting the path of the robot to an approximate straight line. The position after n iterations is calculated as below:

$$(x + iy) = \frac{\left(Rho \times e^{i\emptyset}\right) \times (1 - e^{(n+1) \times i\theta})}{1 - e^{i\theta}} \qquad (16)$$

where Rho is the distance the robot moved after one iteration of the LUT, Ø is the direction moved and θ is the heading of the robot.

3) If the robot does not deviate outside the vertical boundaries during all 100 iterations, then the fitness is calculated as in the step below.
   If the position is outside the vertical boundaries, then the iteration is terminated at its current point, and the fitness calculated at this point:

$$TargetF = 100 - \left(\frac{\sqrt{(0-x)^2 + (1000-y)^2}}{10}\right) \qquad (17)$$

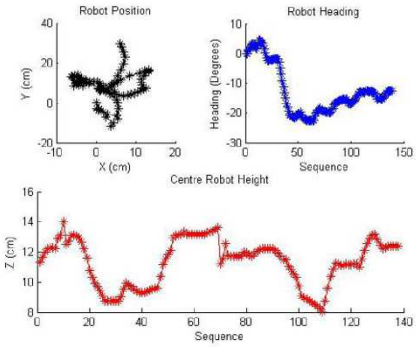


Figure 4. Plot of the 3 movement parameters over a GA generated LUT.

Equation (17) converts the distance from the terminated (x,y) position to the target position (0,1000) to a percentage, which is used as the Target Fitness.

4) The secondary requirements are now calculated; stability and efficiency, as detailed in the following equations:

$$Efficency = 80 - \alpha \times 5 \qquad (18)$$

$$Stability = 50 - \beta \times 8 \qquad (19)$$

where α is the number of steps taken across a single evaluation of the gait, and β is the standard deviation of the height of the robot. Note a minimum of 6 steps has been factored in Equation (18).

5) Finally the total fitness of the gait can be calculated:

$$Final\ Fitness = (Efficiency + Stability) \times 0.25 + TargetF \times 0.75 \qquad (20)$$

A perfect fitness of 100% would result in the robot stopping at exactly (0,1000), with a 6 step tripod gait being evolved, that maintains ultimate stability.

The resulting position across 100 iterations can then be plotted, and an example is shown in Fig. 5.

B. *Genetic Algorithm*

Within this research, the GA is implemented as follows:

1) An initial random population of one hundred LUTs is created using Matlab's random number generation, and using an initial seed created from the current computer time. This allows different LUTs to be generated every time the GA is run.
2) These initial chromosomes are then simulated within the mathematical model, and the fitness value of each is returned.

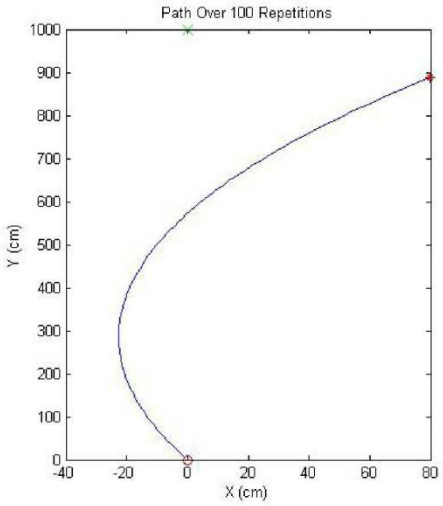


Figure 5. Plot of the robot's position over 100 iterations of a LUT.

3) The population is now expanded, by creating one hundred exact copies of the parents. Each offspring is then procreated using "Two Point Crossover", a method which chooses two random points within a parent's LUT, and copies the contents of LUT between these points into the offspring's LUT. This has the effect of creating an offspring which contains genes from both parent chromosomes.
4) To add diversity to the population, five offspring are randomly chosen to have ten random genes mutated, which is equivalent to a 0.31% mutation rate. This allows the population to maintain diversification so to avoid becoming trapped at local optima of fitness.
5) The offspring chromosomes are now each simulated within the mathematical model, and a fitness value of each is returned.
6) At this point, there are now two hundred chromosomes, and this must be reduced to a stable population size of one hundred. This is performed using "Pair Based Tournament Selection", a process which finds the best individual of a pair of chromosomes; one parent, one offspring. The chromosome with the best fitness is kept, and the worst is discarded. This is performed on one hundred pairs, thus reducing the population size to one hundred.
7) In order to allow variance in pair combinations, the resulting order of the population is now randomised.
8) The best performing chromosome from the selection process is always kept, and its fitness value compared to the desired fitness value. If the best fitness value meets the termination criteria, then the GA is finished, otherwise the GA iterates again from step (3).

The resulting best performing chromosome is now an optimised LUT, which results in an acceptable walking gait of the robot. Within this research there will be multiple acceptable walking gaits for this robot, and the GA has produced several variations.

IV. RESULTS

During the process of this research, the fitness evaluation strategy had to be refined many times over. The first strategy was simply to get the robot to walk as far as possible while maintaining a constant forward facing heading. The result of this was the simulated robot learned to walk diagonally, while maintaining a straight heading.

The second strategy included extrapolating the position of the robot, after *n* iterations of the LUT, and comparing the resulting position to a target location, 1000cm in front of the robot. Under this situation, the robot learned to walk in an arc to the target location, thus also defeating the straight line requirement.

The third strategy including adding vertical boundaries to the extrapolation, with the expectation of limiting the path of the robot. Although the robot would now approach the target position in a straighter path, the evolved gait was very inefficient.

The final strategy, as detailed in Section III, has resulted in the successful evolution of walking gait which results in the simulated robot walking forward in an approximate straight line. Evolution of this gait occurred after 684 generations, and an evolution period of 39 hours.

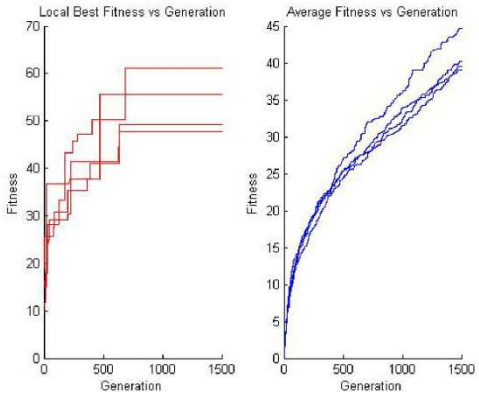


Figure 6. Fitness across four GA evaluations.

The best gait evolved had a final fitness of 61.06%, and followed an approximate straight path, as shown in Fig. 7.

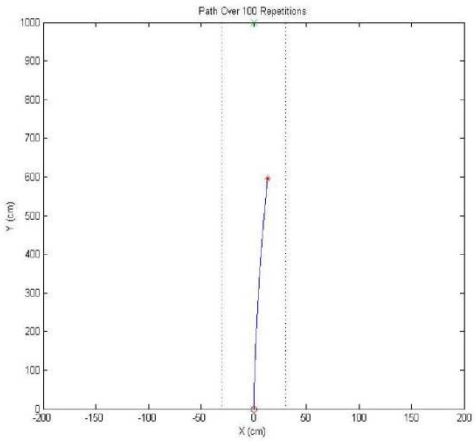


Figure 7. Path of the best evolved gait across 100 iterations.

Although overall the gait resulted in a straight walking path, plotting the movement incrementally showed the gait was typically inefficient. This can be seen in Fig. 8.

The plot of robot position shows the robot dwelling around the centre point, walking forward, then appearing to fall backwards, which can be seen in the robot height plot and corresponding points in the position plot. The stability of the robot was good (approximately level at eleven cm) until the fall happened at sequence iteration one hundred and nine. The plot then shows the robot standing back up to level by the end of the gait.

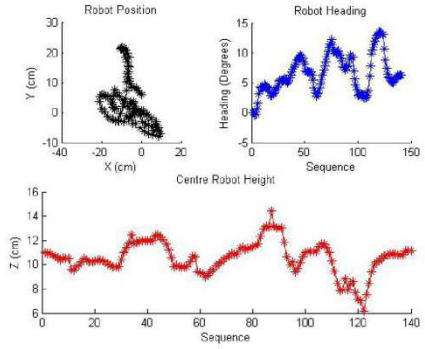


Figure 8. Plot of the movement of the simulated robot .

V. CONCLUSION

The result of this research is that a walking gait has been successfully evolved, which guides the simulated robot in a approximate straight line, however the efficiency of the gait was poor. This is because the fitness evaluation was biased towards forming a gait with a straight line path rather than efficiency of motion.

The GA was run multiple times on multiple PCs with varying rates of successful gait evolution. These fitness values were comparable to the best achieved, however the direction of the path formed larger arcs rather than the desired straight line, although improved efficiency and stability of the gait were exhibited.

REFERENCES

[1] M. Mazzapioda and S. Nolfi, "Synchronization and gait adaptation in evolving hexapod robots," in *Lecture Notes in Computer Science*. vol. 4095/2006: Springer Berlin / Heidelberg, 2006, pp. 113-125.

[2] V. Dürr, J. Schmitz, and H. Cruse, "Behaviour-based modelling of hexapod locomotion: Linking biology and technical application," *Arthropod Structure & Development*, vol. 33, pp. 237-250, 2004.

[3] G. B. Parker, "Evolving leg cycles to produce hexapod gaits," in *The World Automation Congress WAC*, 2000, pp. 250-255.

[4] N. Kohl and P. Stone, "Policy gradient reinforcement learning for fast quadrupedal locomotion," in *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, 2004, pp. 2619-2624.

[5] D. Pratihar, "Evolutionary robotics - a review," *Sadhana*, vol. 28, pp. 999 - 1009, 2003.

[6] G. Parker, D. Braun, and I. Cyliax, "Evolving Hexapod Gaits Using Cyclic Genetic Algorithms," New London, CT: Indiana University.

[7] S. Chernova and M. Veloso, "An evolutionary approach to gait learning for four-legged robots," in *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, 2004, pp. 2562-2567.

[8] D. E. Goldberg, *Genetic algorithms in search, optimisation and machine learning*. MA: Addison-Wesley, 1989.

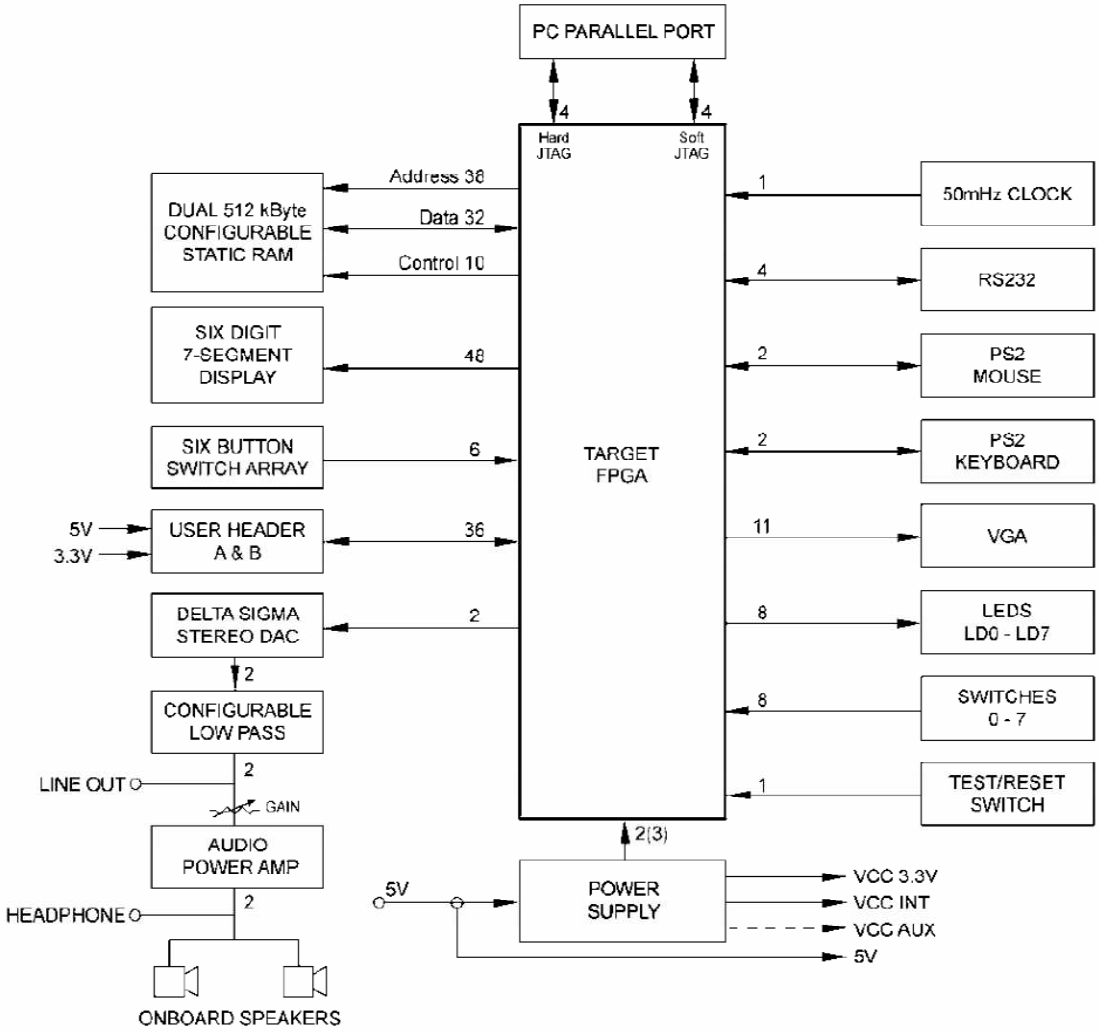
[9] C. Darwin, *Origin of species*. London, UK: John Murray, 1859.

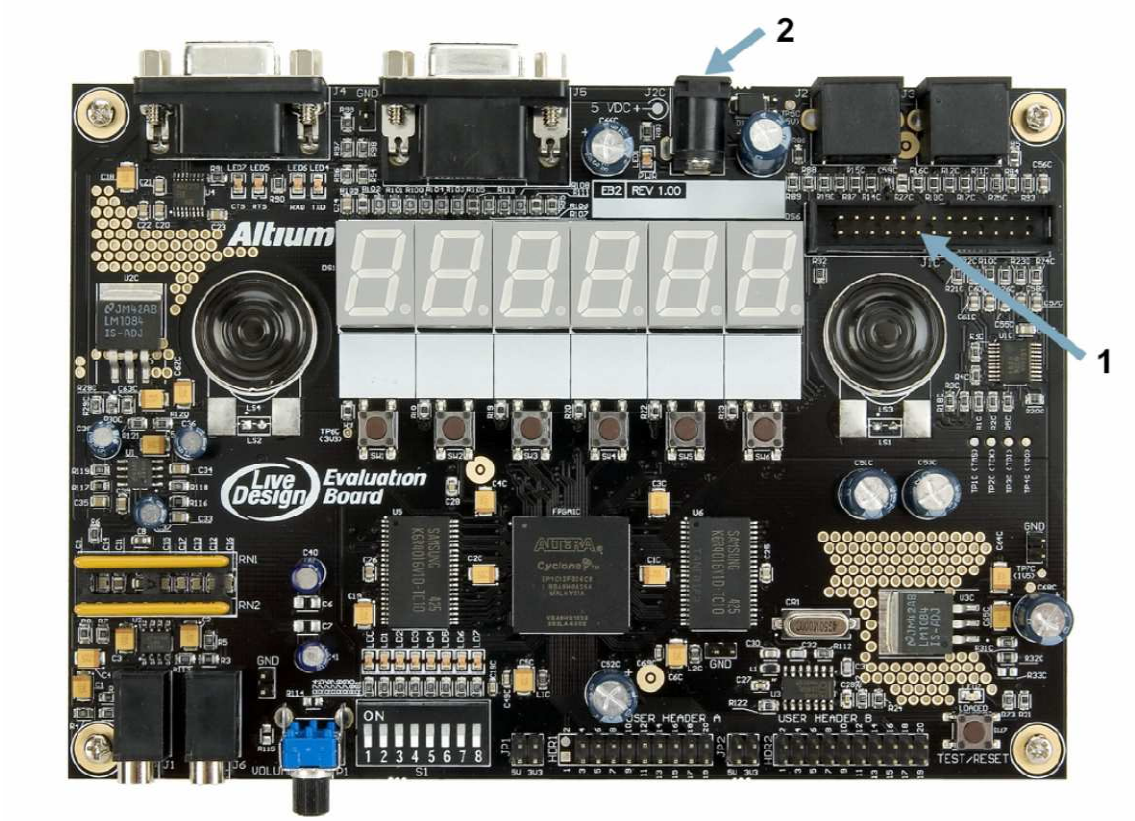
[10] E. Alba and F. Chicano, "Genetic Algorithms," in *Metaheuristic procedures for training neutral networks*. vol. 36: Springer US, 2006, pp. 109-137.

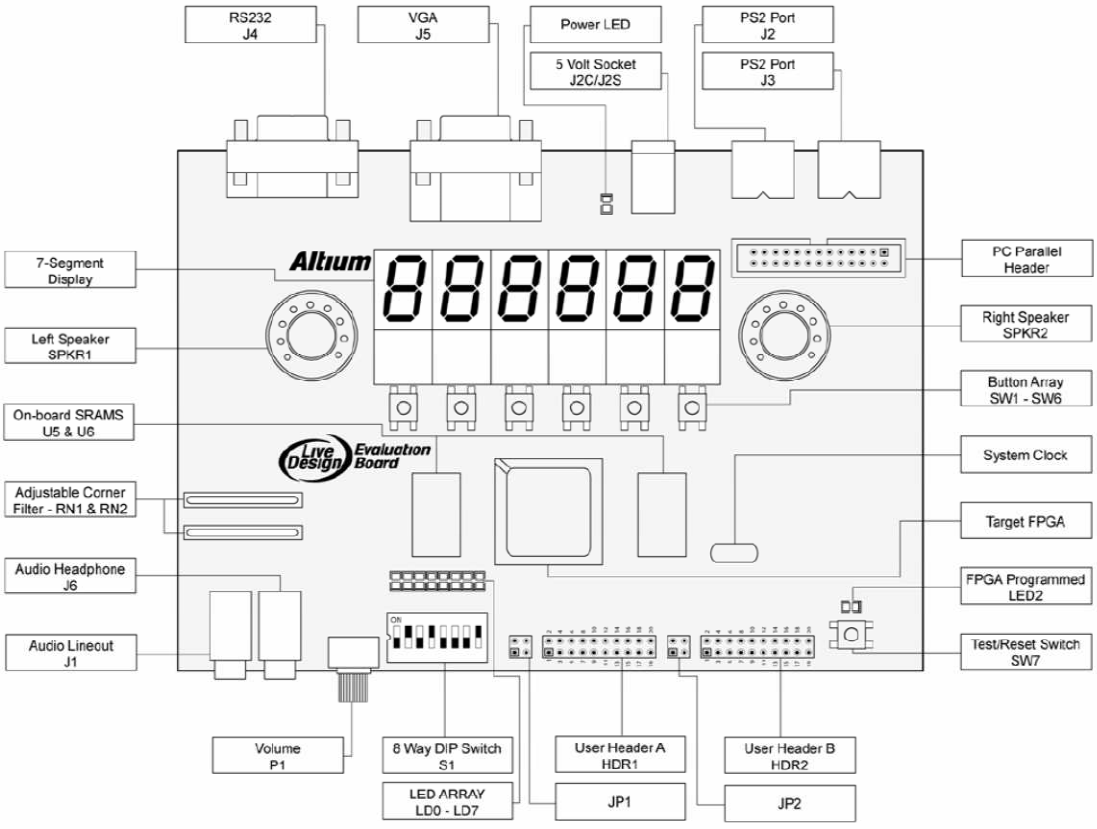
[11] T. Braunl, "Genetic Algorithms," in *Embedded Robotics*: Springer Berlin Heidelberg, 2006, pp. 291-305.

[12] J. Fauvel, *Mobius and his Band: Mathematics and Astronomy in Nineteenth-Century Germany*: Oxford University Press, USA, August 1993.

Appendix B Altium Live Design Board used for Experimentation




2
1
Altium
Live Design
Evaluation Board
REV 1.00
5 VDC
USER HEADER A
USER HEADER B
TEST/RESET
ON
1 2 3 4 5 6 7 8


RS232
J4
VGA
J5
Power LED
5 Volt Socket
J2C/J2S
PS2 Port
J2
PS2 Port
J3
7-Segment
Display
Left Speaker
SPKR1
On-board SRAMS
U5 & U6
Adjustable Corner
Filter - RN1 & RN2
Audio Headphone
J6
Audio Lineout
J1
PC Parallel
Header
Right Speaker
SPKR2
Button Array
SW1 - SW6
System Clock
Target FPGA
FPGA Programmed
LED2
Test/Reset Switch
SW7
Volume
P1
8 Way DIP Switch
S1
LED ARRAY
LD0 - LD7
User Header A
HDR1
JP1
User Header B
HDR2
JP2
Altium
Live Design
Evaluation Board