Real Time Evolutionary Algorithms in Robotic Neural Control Systems

This thesis is submitted to

The Robert Gordon University

in partial fulfilment of the requirements for

the degree of Doctor of Philosophy

Ananda Prasanna Jagadeesan

School of Engineering
The Robert Gordon University
Aberdeen, Scotland.

2006

Declaration

I hereby declare that this thesis is a record of work undertaken by myself, that it has not been the subject of any previous application for a degree and that all sources of information have been duly acknowledged.

Ananda Prasanna Jagadeesan

2006

Abstract

This thesis describes the use of a Real-Time Evolutionary Algorithm (RTEA) to optimise an Artificial Neural Network (ANN) on-line (in this context "on-line" means while it is in use). Traditionally, Evolutionary Algorithms (Genetic Algorithms, Evolutionary Strategies and Evolutionary Programming) have been used to train networks before use - that is "off-line," as have other learning systems like Back-Propagation and Simulated Annealing. However, this means that the network cannot react to new situations (which were not in its original training set).

The system outlined here uses a Simulated Legged Robot as a test-bed and allows it to adapt to a changing Fitness function. An example of this in reality would be a robot walking from a solid surface onto an unknown surface (which might be, for example, rock or sand) while optimising its controlling network in real-time, to adjust its locomotive gait, accordingly.

The project initially developed a Central Pattern Generator (CPG) for a Bipedal Robot and used this to explore the basic characteristics of RTEA. The system was then developed to operate on a Quadruped Robot and a test regime set up which provided thousands of real-environment like situations to test the RTEA's ability to control the robot. The programming for the system was done using Borland C++ Builder and no commercial simulation software was used. Through this means, the Evolutionary Operators of the RTEA were examined and their real-time performance evaluated.

The results demonstrate that a RTEA can be used successfully to optimise an ANN in real-time. They also show the importance of Neural Functionality and Network Topology in such systems and new models of both neurons and networks were developed as part of the project. Finally, recommendations for a working system are given and other applications reviewed.

Acknowledgement

I would like to thank my father Mr. Jagadeesan Ramasamy and mother Mrs Lakshmi Prabha Jagadeesan for the love and support they have given me. Also would like to thank my sister Bhavani and her family for their support.

I am indebted to both of my supervisors, Mr Grant Maxwell and Dr. Christopher Macleod for their unrelenting assistance and advice. Both of my supervisors have been my inspiration and I consider myself to be very lucky to have been their student. I am also indebted to Mrs Ann Reddipogu, for her support and advice.

I would also like to express my gratitude to, my friends Mr Balakrishnan Shanmugam, Miss Hema Narasimhan, Mr Samarasena Buchala, Miss Aurélie Guyard, Mr Robert Welbourn, Mr Shahryar Iqbal, Mr Gideon Damaryam, Mr Sethuraman Muthuraman, Mr Alagappan Viswanathan, Mrs Anita Teo, Hazel and Mrs Shayne Reed for their support.

I am also grateful for the encouragement during this project from The Robert Gordon University, School of Engineering staff.

Ananda Prasanna Jagadeesan

Contents Table

Chapter 1

Introduction

1.1 Introduction to Chapter

This chapter gives a brief overview of the project. It starts by explaining the motivation of the research. This is followed by a discussion of the aims and objectives of the work. An outline of the innovative ideas discovered during the project is then given. Finally, a breakdown of the thesis chapters is included.

1.2 Introduction to Project

Since the middle of the 20th century, scientists have been studying biological intelligence in order to try to create Artificial Intelligence (AI). It is thought that AI will add considerably to the functionality of computers, robots and other machines; however, the complexity of the nervous system makes it one of the biggest challenges that mankind has ever faced.

One approach to AI is biologically inspired Artificial Neural Networks (ANNs). These are computer or electronic models of biological nerve cells or neurons; they are often simulated by software [1]. A major advantage of the ANN is that it can improve its performance using a training or learning algorithm.

One of the main engineering applications of ANNs is in Artificial Neural Control [2]. This involves using an ANN as a system controller instead of traditional methods (for example, a PID controller or compensator). There are many advantages of doing this, including the ANN's ability to learn and its performance in non-linear systems. In this project, an ANN is used to control a legged robot. ANNs used to generate the rhythmical leg patterns for walking in such robots are often called Central Pattern Generators (CPGs).

ANNs are normally trained before use, off-line. On-line learning has proved difficult [3]. The disadvantage of this may be seen by considering a legged robot walking across a series of different ground types (for example, sandy, rocky or boggy). Obviously, in this case, for optimum locomotive efficiency, the robot will have to alter its gait pattern in response to the conditions underfoot. It may have to shorten its stride, for example, when moving from a hard to a sandy surface. Such parameter changes are difficult unless there is a method of training the ANN as it is being used.

One method of training an ANN is to use an Evolutionary Algorithm (EA). Such algorithms are based on biological evolution and allow the ANN to evolve to a good solution. However, EAs are normally also used off-line.

The idea behind this project is to use a Real-Time Evolutionary Algorithm (RTEA) to constantly train the robot's controlling neural network, evaluating changes to its fitness function (in the first approximation, the efficiency of locomotion). In this way, the robot's control system is constantly seeking a better solution and will move towards such a solution even when the robot moves onto a different surface.

Since Artificial Neural Networks are used to control the robot's legs, rather than direct control (for example, an algorithm generating a rhythmical step pattern), lessons learned from the experience can be applied more generally to other neural network controlled systems. Such networks are commonly used to control a variety of different mechanical and mechatronic systems.

1.3 Aim and Objectives

The aim of this research is to investigate the use of Real-Time Evolutionary Algorithms in on-line training of Neural Control Systems. The project uses a simulated legged robot as an experimental test-bed.

The following objectives were set out at the beginning of the project.

Background Reading and Appropriate Directed Study

Appropriate directed studies were undertaken at the beginning of the research. These included attending seminars, lectures, doing practical exercises in the field of study and reproducing the work done by McMinn (a previous researcher in the group).

Literature Search in Field

The previous work of the group and in particular, McMinn's [4] Central Pattern Generator (CPG) Artificial Neural Network was studied to understand its relevance to the project. A wide range of techniques related to Real-Time Evolutionary Algorithms were selected and studied carefully. The main topics covered included Genetic Algorithms, Evolutionary Programming, Evolutionary Strategies, Simulated Annealing and their use in control systems.

Development of a Central Pattern Generator (CPG) Artificial Neural Network for a Bipedal Walking Robot

Initially, McMinn's neuron model [5] was tested by training it using a GA and its attributes were investigated. Then a new neuron model was developed using a simulated biped robot as a test bed. This involved developing the biped model and environment to simulate walking patterns in real time.

Investigation of Evolutionary Algorithms to train the CPG Artificial Neural Network

Initially it was planned to test the system using different off-line EAs but, as the work went better than expected, research was begun on the RTEA immediately. An investigation was carried out to learn more about the mutation and other evolutionary operators. As a result, optimal parameters for the basic RTEA were selected and tested. Finally, an initial implementation of a RTEA was tested on the system. Further details of this investigation are discussed in chapter 6.

Comparison with previously obtained results

The test results of the initial RTEA were compared with the results obtained in previous work [6].

Extension of MPhil work from Biped to Quadruped robot

The work was taken from the MPhil to the PhD stage by extending it from a Biped robot to a Quadruped. This involved the transformation of single degree of freedom actuators per leg in the biped robot, into two degree of freedom actuators per leg in the quadruped. As the number of legs and degrees of freedom are increased, the number of neurons in the CPG network also increased. A deeper investigation of the CPG network resulted in different neuron models. These models were analysed and tested. From the tests, the best network was selected for the quadruped robot. Further details about the neuron model development are given in chapter 6.

Extension of Evolutionary Algorithm to Real Time Evolutionary Algorithm to train the CPG artificial neural network for a walking robot in any gait
The initial implementation of the RTEA was developed further to adapt it to optimise the robot in any gait. This was achieved by investigating the different ways of implementing the RTEA – for example, different mutation and other operators. Performance (fitness) was measured based on the distance covered, stability and the fuel efficiency of the robot. The RTEAs and their operators are discussed in chapter 7.

Selection of best algorithm from the real time evolutionary algorithms developed
The selection of the best RTEA was based on the "Success Rate" and the "Fitness" of the system. Success Rate was the measure of how fast and efficiently the RTEA evolved the CPG of the quadruped robot to walk and the Fitness was the level of adaptation to the environment. Various experiments were conducted on the RTEA and the best algorithm was selected.

Running and testing the best algorithm
The system was tested further and its results were analysed. The results are discussed in detail in chapter 8.

Comparison with published benchmarks and results from other researchers
The results from the best algorithm were compared with published results.

1.4 Innovative Aspects of Research

The project makes several new contributions to "the art." Some of its most important findings are listed below:

- Investigation of several of the commonly used neuron models revealed that when networks have to undergo an evaluated gradual dynamic change (as is the case in the RTEA), neuron models using threshold functions are inappropriate. The reasoning behind this is explained in chapter 6. This finding led to the development of new models appropriate for use in the system.

- The topology of the network as a whole was also investigated as part of the project. In a similar way to the findings for the neuron model, it was found that certain topologies worked better than others in the system. Again, this led to the development and testing of appropriate neural network structures for the task at hand.
- From the two points above, it may be seen that the network needs to be designed carefully if it is to be used with the RTEA. This led to a consideration of the types of system suitable for the approach outlined in this thesis (and conversely which systems are not appropriate).
- Finally, and most importantly, the implementation and extensive investigation of a Real - Time Evolutionary Algorithm controlling a Neural Net is a unique piece of work.

1.5 Structure of Thesis

Chapter 2: Previous Works in Group

This chapter describes the work done by previous workers within the research group and shows the development and context of current work.

Chapter 3: Evolutionary Algorithms

This chapter explains various traditional Evolutionary Algorithms and discusses the advantages and disadvantages of each of them.

Chapter 4: The Real Time Evolutionary Algorithm

Based on the discussion in the previous chapter, this chapter explains the development of the Real-Time Evolutionary Algorithm.

Chapter 5: Literature Review

This chapter contains the results of a comprehensive literature review in the area of study.

Chapter 6: Investigation of Evolutionary Algorithms and System Implementations
The chapter explains how the research led to the development of neuron and network models especially suited to RTEAs. It also describes the initial investigation of the RTEA in a biped robot and the development of the robotic models used.

Chapter 7: From Bipeds to Quadrupeds
This chapter explains the development of quadruped robot models, using what was learned from the biped robot simulators and neural networks.

Chapter 8: RTEA Operators
This chapter explains the different RTEA parameters and their roles in controlling the network performance. It also explains the RTEA's operators and their advantages and disadvantages are discussed using appropriate performance graphs.

Chapter 9: Future Work
In this chapter suggestions are made for further work.

Chapter 10: Conclusion
This chapter evaluates the research and discusses its successes and failures in terms of the original objectives.

Published papers, reports produced during the course of the research, and further results are included in the appendices.

Chapter 2

Background and Previous work within the Research Group

2.1 Introduction

Since its foundation in 1994, the Artificial Neural Networks group at The Robert Gordon University has gained a considerable amount of experience in the field of Evolutionary Artificial Neural Networks. This chapter explains the background of the group's work and describes how this project evolved from the previous research within the group. It starts by presenting the basic features of the biological nervous system and describes the artificial equivalent. It then explains the group's work on artificial nervous systems. Finally, it explains this project's origins.

2.2 Biological Nervous System

Nerve cells or neurons are the basic building blocks of Biological Nervous Systems [1] (BNSs). In higher animals, billions of nerve cells work together to form a large Neural Network. This neural network performs complex and organized communication and information processing within the nervous system and between it and the external environment. The main task of the nervous system is to ensure that the organism reacts optimally to its environment.

Anatomically the nervous system can be divided into the "Central Nervous System" (CNS) and "Peripheral Nervous System" (PNS). The CNS consists of the brain and spinal cord and the PNS consists of the sensory and motor neurons which connect the outside world to the CNS. Higher organisms (like reptiles and mammals) generally have more complicated nervous systems than simpler organisms.

The nervous system has sensory neurons which receive information from the outside world. These neurons receive stimuli (sensory information) and send it to the central nervous system in the form of nerve impulses. After processing in CNS, information is relayed to other neurons (called motor neurons) which carry out the body’s actions via muscles and glands [2].

2.3 Biological Neural Networks

In 1840 anatomists Jacob Schleiden and Theodor Schwann showed cells are the basic units of tissue architecture. Later in the 19th century neuro-anatomists Santiago Ramón y Cajal and Camillo Golgi said that there are large numbers of discrete cells in the brain [3]. These cells are known as neurons which are the elementary unit of processing information (different activities of body) within the Biological Neural Network (BNN). Hence the BNN consists of neurons connected to each other. The figure below shows a representation of a neuron (Fig 2.1).

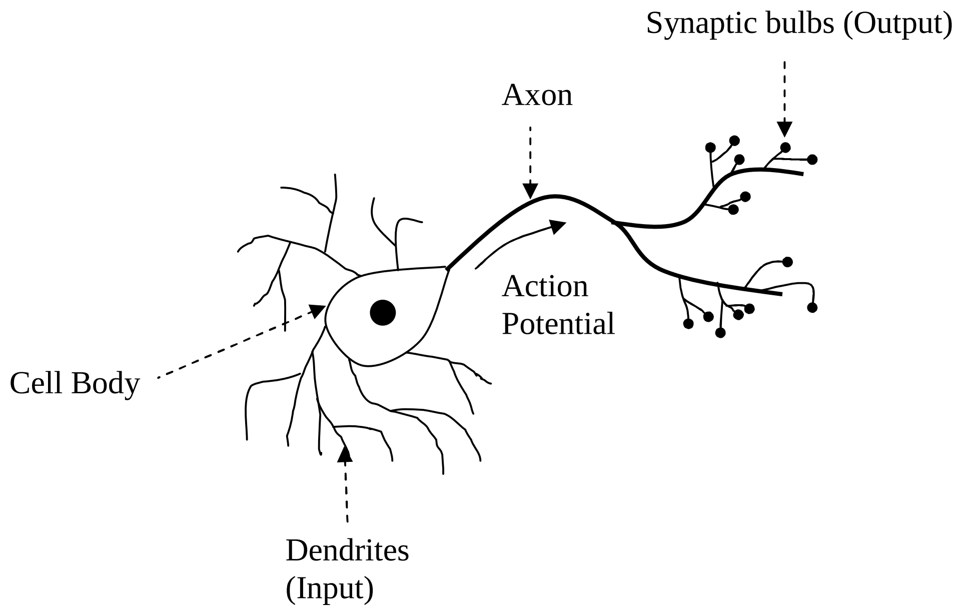


Figure 2.1: A Biological Neuron

As shown in the figure, the neuron has a “Cell body” (also called a soma), which is the main information processing centre. The input sources for signals to be fed to the cell body are the “Dendrites”. The dendrites receive nerve pulses from other neurons or from sensory input to the body. When the received information from all these dendrites is strong enough to overcome the neuron’s threshold, the information processing takes place within the cell body as an ionic chemical reaction and this results in a pulse. This pulse is called an “Action Potential”; it travels down the axon. The figure 2.2 shows a graph of action potential voltage against time. In this example, the neuron’s threshold value is -55 mV.

The junction between the axon and the dendrite is called a synapse. There are two types of synapse: an Electrical Synapse and a Chemical Synapse. Electrical Synapses conduct the signal directly by contact. Chemical Synapses have a gap between the pre-synaptic and postsynaptic terminals of the neuron. A neuroactive substance transfers the signal from the pre-synaptic to the postsynaptic neuron and is called a neurotransmitter.

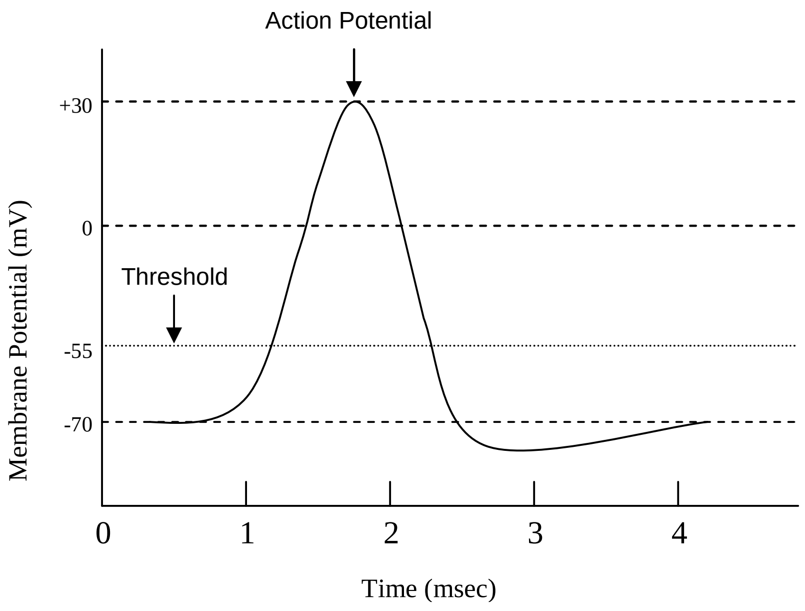
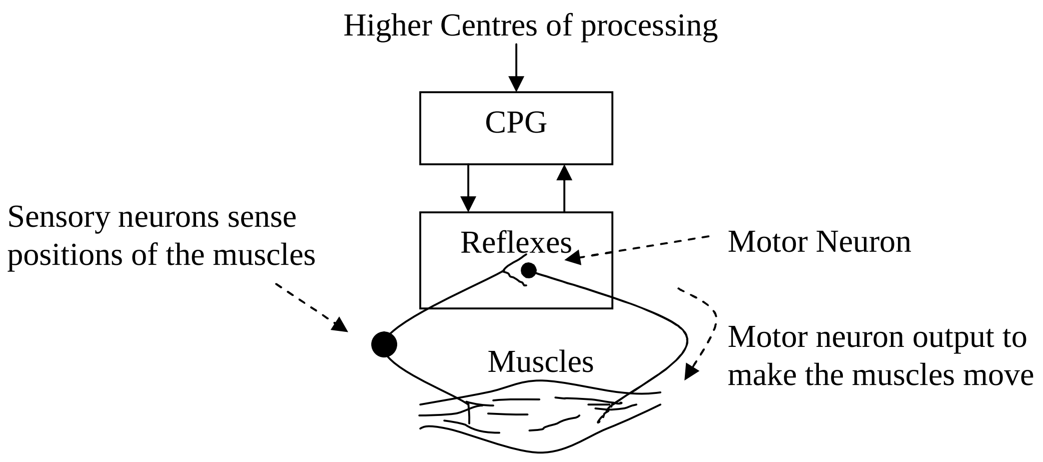


Figure 2.2: Action Potential

The strength of the transferred signal depends upon the physical and chemical characteristics of the synapse. By adjusting the amount and type of the neurotransmitter, the stimulation of the neuron can be adjusted. This mechanism is critical in learning [4].

2.4 Components of Locomotion

Since this project uses a legged robot as a test bed, it is appropriate at this stage to consider that part of the BNS which deals with locomotion. The locomotory functions of the BNS may be divided up into two main sections. The Central Pattern Generator (CPG) and the Reflex system, as shown in figure 2.3. Each of these functions is explained below.



Higher Centres of processing
CPG
Reflexes
Sensory neurons sense positions of the muscles
Motor Neuron
Motor neuron output to make the muscles move
Muscles

Figure 2.3. The Locomotory Components of the BNS

2.4.1 Reflex

The Reflex is a Control System [5]. The input is a stimulus either from the higher brain centres (voluntary movement) or from within the spinal cord (involuntary movement). There are many kinds of reflexes (e.g stretch reflex, withdrawal reflex, etc), but most of them share the same basic functions. The reflex is mediated by the "Reflex Arc" (Reflex pathway). The Reflex Arc is the structural basis of a reflex, as explained below [6].

1. A stimulus is received by a receptor and converted into an action potential.
2. The sensory neuron conducts the action potential to the central nervous system.
3. The signal is modified (to provoke the necessary output) by other neurons in the reflex centre. Signals are then passed to effectors (which may be muscles or glands), by motor neurons.
4. The signals passed to the organs produce corresponding action(s).

Note the similarity of the reflex to a traditional control system, the inputs (set points) corresponding to input from the higher centres and the sensory neuron providing feedback.

2.4.2 Central Pattern Generator (CPG)

The CPG generates signals which control repetitive rhythmical patterns like walking, running and flying. Grillner and Wallén [7] showed that neural circuits in the spinal cord are responsible for generating the rhythmical motor patterns which cause locomotory movement. They also proved that these movements can happen with or without inputs from the higher brain centres.

For a walking animal, the CPG selects an appropriate gait based on its circumstances. This depends on the strength and type of the stimulus received from the higher brain layers. The CPG layer may also receive information from the reflex layer, and does not require input from the higher brain centres.

2.5 Artificial Nervous System

2.5.1 Structure of ANS

In 1998 MacLeod, McMinn and Maxwell [8] proposed a biologically inspired framework for an Artificial Nervous System (ANS). Figure 2.4 shows the hierarchical structure of the ANS. The layers marked in asterisk are those which can have multiple modules in parallel.

2.5.2 Layers of the ANS

The layers of the ANS can be divided into five sections. They are explained below.

1. The lowest layer of the ANS is the reflex layer. The reflex layer is similar to the reflex arc of the BNS and is responsible for the simplest controlled forms of movement, interfacing directly with the hardware actuators of the animat. There can be multiple reflex module layers in parallel, each controlling different actuators.

2. The action layer is responsible for synchronising the reflexes together and producing rhythmic movements of the body; for example, walking, running and swimming. These actions correspond to the Central Pattern Generators (CPGs) mentioned above. This layer can also semi-autonomously control the actions and reflexes - for example, taking a wide stride to avoid an obstacle.

3. Animals have different stereotypical behaviours like eating and mating. Different actions and reflexes are sequenced together to produce such behaviours in the behaviour layer.

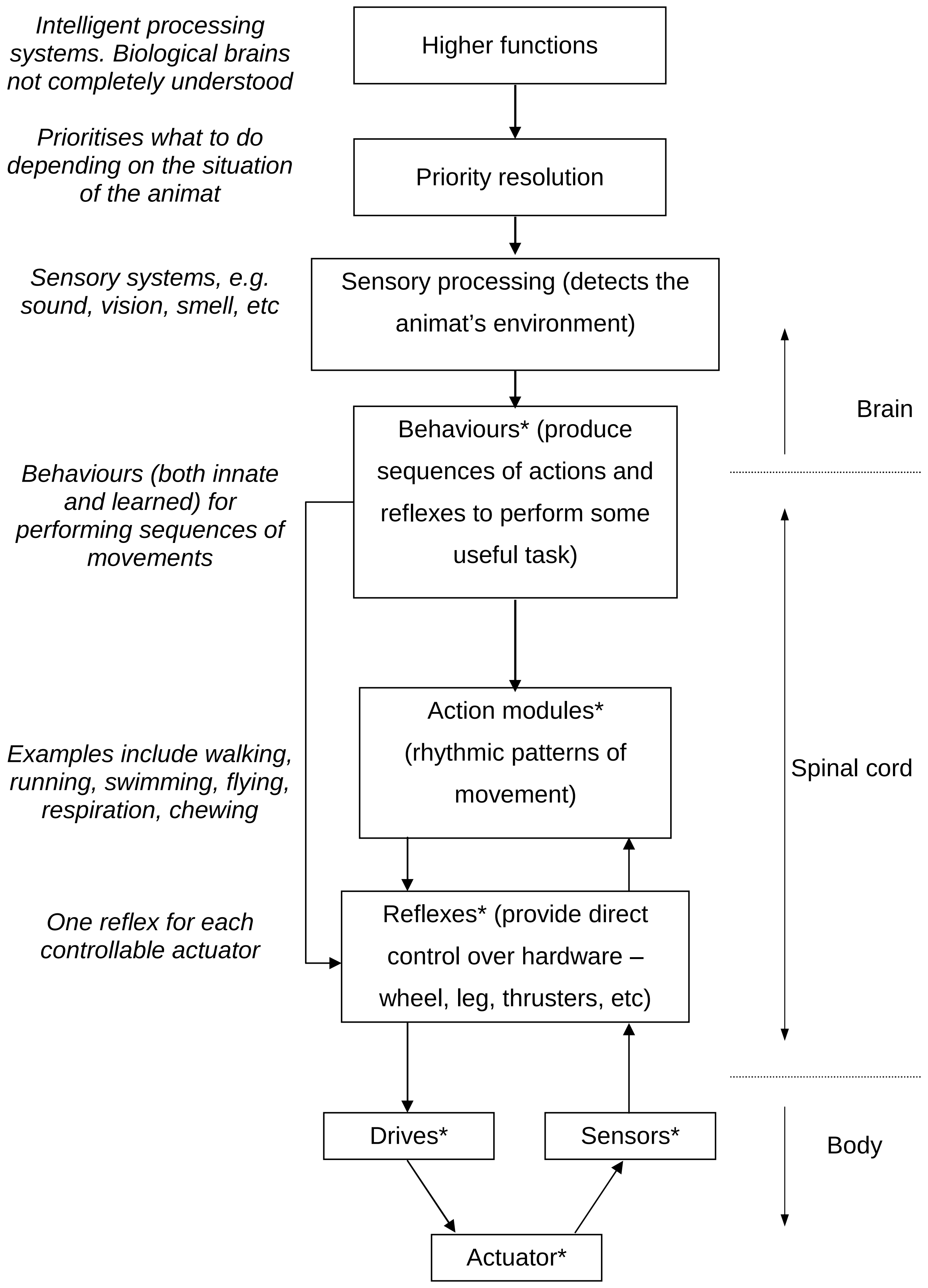

Intelligent processing systems. Biological brains not completely understood
Higher functions
Prioritises what to do depending on the situation of the animat
Priority resolution
Sensory systems, e.g. sound, vision, smell, etc
Sensory processing (detects the animat's environment)
Brain
Behaviours (both innate and learned) for performing sequences of movements
Behaviours* (produce sequences of actions and reflexes to perform some useful task)
Examples include walking, running, swimming, flying, respiration, chewing
Action modules* (rhythmic patterns of movement)
Spinal cord
One reflex for each controllable actuator
Reflexes* (provide direct control over hardware – wheel, leg, thrusters, etc)
Drives*
Sensors*
Body
Actuator*

Figure 2.4 Artificial Nervous System Structure

(Reproduced by permission of McMinn)

4. The survival of an organism depends upon its sensory inputs - for example, smell, vision and hearing. Such sensory input is processed by the sensory processing layer.

5. The priority resolution layer makes decisions depending upon the situation and internal status of the animat. This is connected to higher layers of ANS. The purpose of this layer is to resolve any conflicts between behaviours (for example, escape in the presence of a danger or move towards food).

2.5.3 Advantage of ANS

The Modularity of the ANS is its main advantage. Modules can be added to any layer to perform new tasks. The ANS may be implemented using any conventional technology, so it is a general system and not dependent on any particular implementation. If the hardware implementation requires a change in a lower level, then only that layer need be subjected to change and the higher levels will remain the same. The ANS can be applied in ROVs (Remotely Operated Vehicles), robots, aircraft, or any other system requiring intelligent control.

2.6 Previous Researchers' Contributions to ANS

2.6.1 Evolution of Animat Nervous System (ANS) – Lower Layers

Initially McMinn upgraded the original ANS structure by altering the flow of information (between the ANS layers) from unidirectional to bidirectional. Hence if the ANS encounters a new environment, it can use the higher functions to prioritise the current conditions and initiate the necessary behaviour to tackle that condition. With the new ANS (figure 2.4) structure as a basis, McMinn developed an Evolutionary ANN and implemented the Central Pattern Generator (Action Layer) and Reflex (Reflex Layer) for robot locomotion [5]. Figure 2.5 shows a block diagram of the functionality of McMinn's artificial reflex.

The reflex ANN circuitry receives information from higher layers and actuator sensors. According to the received information, the ANN controls the position of the actuators. A simulation of DC electric motor powered legs was used to test the system.

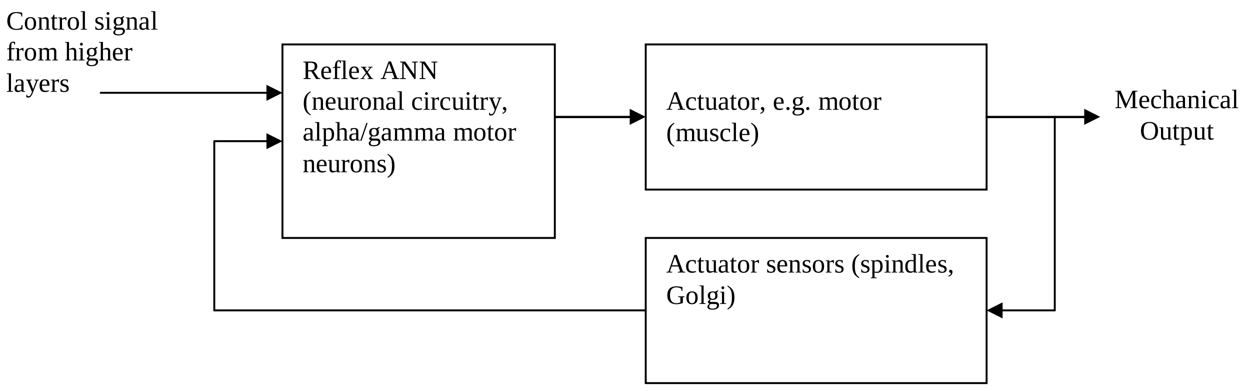

Control signal from higher layers
Reflex ANN (neuronal circuitry, alpha/gamma motor neurons)
Actuator, e.g. motor (muscle)
Mechanical Output
Actuator sensors (spindles, Golgi)

Figure 2.5 Functional block diagram of artificial stretch reflex, with biological equivalent parts in brackets (Reproduced by permission of McMinn)

McMinn used simple network topologies (feed-forward and recurrent networks) in his implementations. The network used a McCulloch-Pitts neuron model and was trained with Evolutionary Algorithms (EAs).

McMinn then constructed the action layer that would sequence the reflexes together and successfully evolved CPGs for biped and quadruped gaits using the system.

It was found that the simple McCulloch-Pitts neuron did not produce useful results and hence a new neuron model was developed that produced timings required for the CPGs. As shown in figure 2.6, the output of the evolved CPG was passed to the Reflex through a leaky integrator. The leaky integrator converted a time domain input to a continuous output value. More information about the time dependent neuron model can be found in [5].

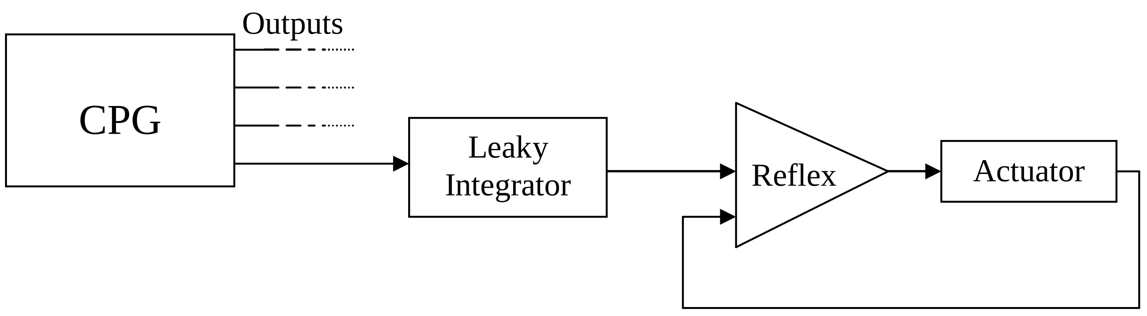


Figure 2.6 Chain of connections from CPG to robot actuator

(Reproduced by permission of McMinn)

McMinn investigated further the structuring of the network. An evolved biped CPG was used as an oscillator, which feeds a pattern generator producing quadruped gaits like Gallop, Trot, Pronk, and Walk. Figure 2.7 shows this alternate setup. Figure 2.8 shows the output of an evolved quadruped gallop.

A conclusion of McMinn's experiments was the suggestion that more modular CPGs are easier to evolve.

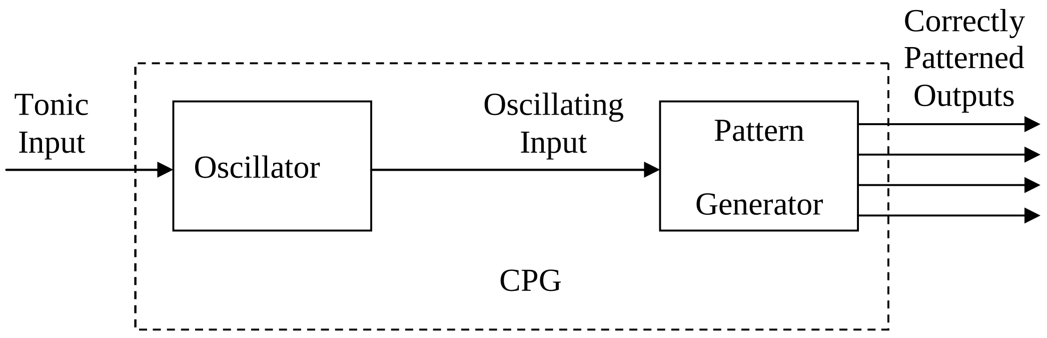


Figure 2.7 Connectivity of the functional units in alternate CPG strategy

(Reproduced by permission of McMinn)

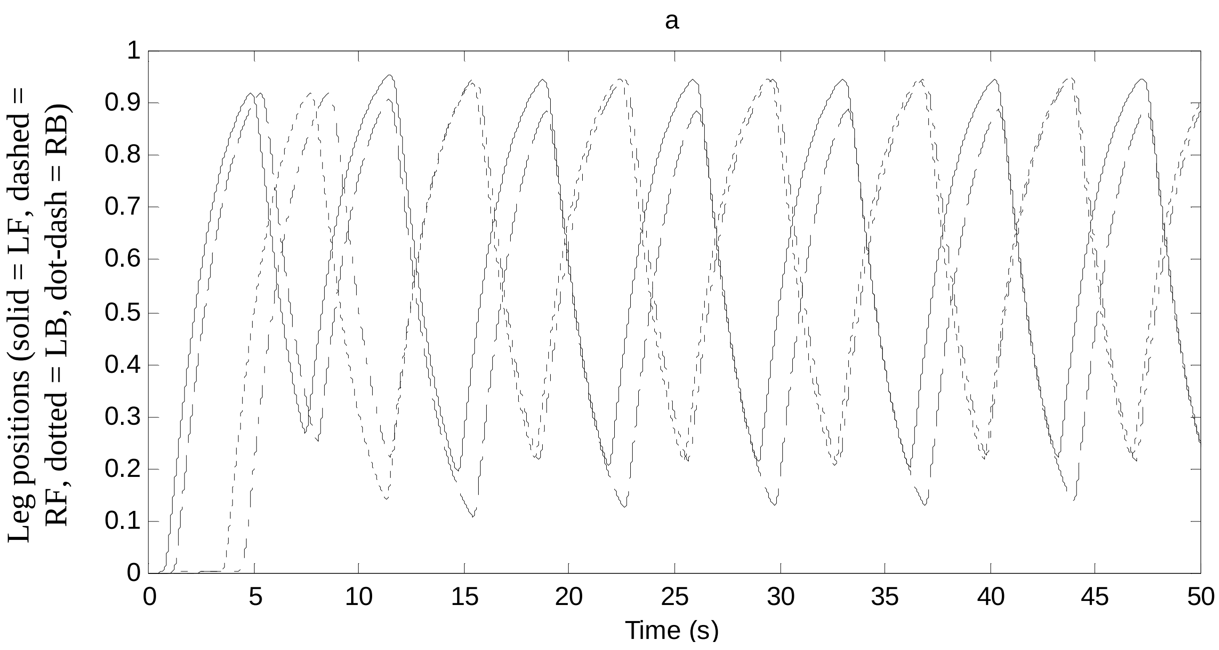


Figure 2.8 Robot leg positions for Quadruped gallop with split CPG

(Reproduced by permission of McMinn)

2.6.2 Evolution of Animat Nervous System (ANS) – Upper Layers

Reddipogu [9] concentrated on the sensory (upper) layers and particularly the processing of visual information. Reddipogu used the toad's visual system as a model for her research. This was because it had a simple neural network structure and hence it was more straightforward to implement an artificial equivalent of it. Reddipogu developed a biologically inspired vision system (based on this) which has the ability to differentiate between prey and predator.

The resulting neural network is shown in figure 2.9. It was trained using an Evolutionary Algorithm based on Reinforcement Learning (EARL).

The trained network was tested using new patterns and its performance was investigated. Figure 2.10 shows a typical output of the network. The horizontal axis represents the classes of outputs and the vertical axis corresponds to the activation level of each output neuron. The terms prey, predator, snap and orient refer to the toad's biological behaviour.

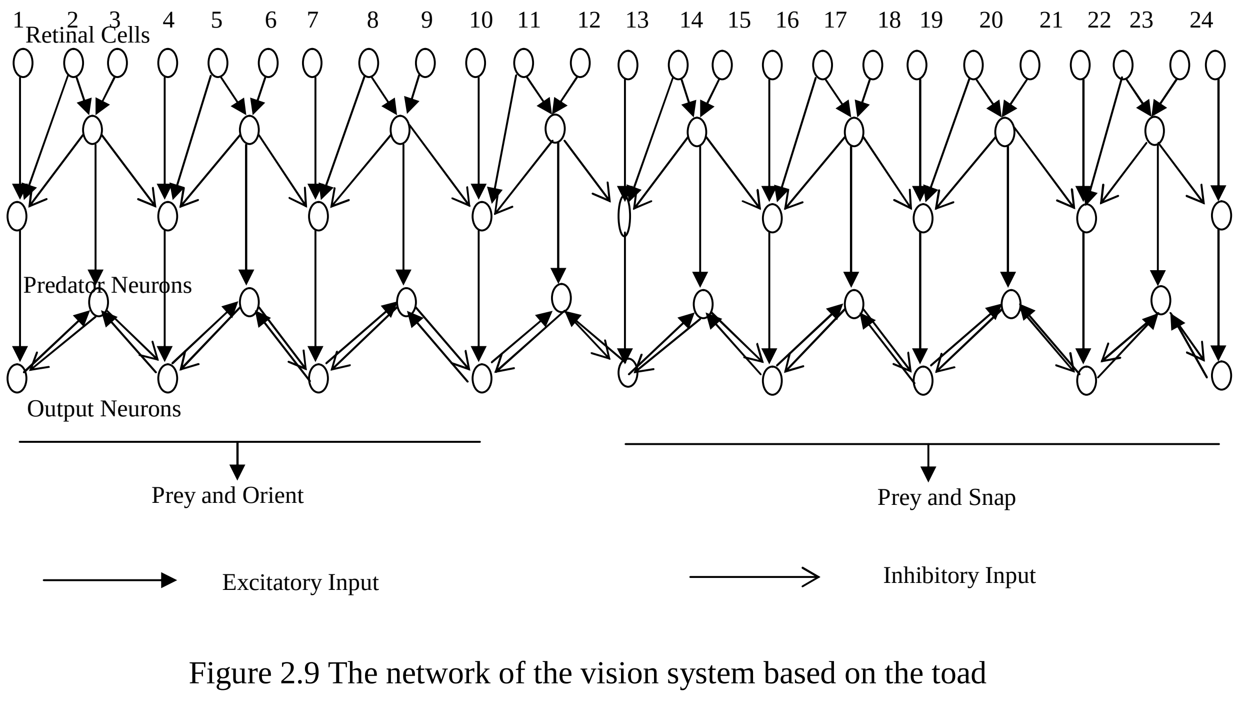


Figure 2.9 The network of the vision system based on the toad

(Reproduced by permission of Reddipogu)

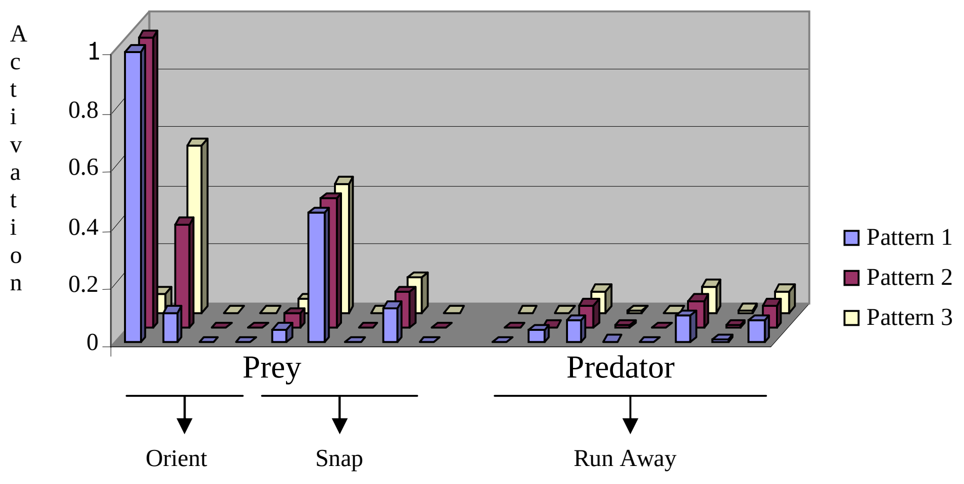


Figure 2.10 Output for Prey and Orient input pattern

(Reproduced by permission of Reddipogu)

Reddipogu showed that the network was able to recognize similar patterns. The results showed that a robotic vision system can be implemented based on these ideas. It also showed, like McMinn, that the performance of the network depends on its modularity. Further detailed analysis of this network can be found in [9].

2.7 Next level of Contribution to ANS

McMinn and Reddipogu's work was aimed at investigating the effect of modularity on the network and its evolution. However, it should be noted that all networks involved in their work needed pre-training before they were implemented in the system. This means that they cannot handle events other than those they were trained for.

For example, McMinn developed a Central Pattern Generator (CPG) using an Evolutionary Artificial Neural Network, which allows a robot to walk with a specific gait. The system does not allow the gait to alter (for example, in terms of stride length or speed) to accommodate differing environment conditions (such as ground softness or substrate). This was seen as a disadvantage, both theoretically and later during the course of actual experiments. The project described in this thesis is a continuation and development of McMinn's work. It allows a real time evolutionary algorithm to control the robot, so that it can change gaits to accommodate conditions underfoot as it is walking.

2.8 Summary

The research group at the School of Engineering at the Robert Gordon University has done extensive work in the field of Evolutionary Artificial Neural Networks and their applications to Robotics. This work has included the implementation of sophisticated Artificial Nervous System-based Robotic Controllers.

It was found during experimentation with such controllers that their off-line pre-use training was unsatisfactory and that some way was needed of training or optimising their performance on-line. It was therefore decided that a project should be initiated to investigate whether this could be done using Real-Time (on-line) Evolutionary Algorithms. This, then, is the basis of the research presented in this thesis.

The next chapter will outline the Evolutionary Algorithms available for use in the development of such a system.

Chapter 3

Evolutionary Algorithms

3.1 Introduction

Evolutionary Algorithms (EAs) are some of the important optimization techniques to emerge in the last three decades. EAs were originally inspired by biological evolution; however, several other fields including mathematics, biology and thermo-dynamics have also contributed to their development. Biological Evolution optimizes the characteristics of different species to fit into a dynamic environment. Haupt, in their book [1], have summarised Grant's [2] explanation about the relationship between optimization and evolution:

> *"Imagine the organisms of today's world as being the result of many iterations in a grand* ***optimization algorithm****. The cost function measures survivability, which we wish to maximize. Thus, the characteristics of the organisms of the natural world fit into this topological landscape."*

In a similar fashion, the EAs optimize the parameters of a given problem, to produce a good result. Each individual EA uses its own operators to find the optimal solution for a given problem. The most important EAs are: the Genetic Algorithm (GA), Evolutionary Strategy (ES) and Evolutionary Programming (EP). Another algorithm, Simulated Annealing (SA) is another type of optimization technique, which is inspired by the annealing process of a hot substance to form a crystalline lattice. Although not normally considered as an EA, SA bears some similarities to the EAs mentioned above because it uses a 'mutation-like' operator to change its parameter gradually and it will be considered in this thesis.

This chapter starts with a brief introduction to the background of the EAs and follows with sections explaining the different EAs and their operators.

3.2 Biological Optimization

As mentioned earlier, biological evolution has been the main inspiration behind most of the EA techniques. Hence, in order to understand the basic concept of the EAs, this section first gives an overview of biological evolution.

To understand evolution, the explanation is started from the cellular level. Every organism inherits its qualities from its parents through genes. Genes hold information (eye colour, hair colour, etc) about the organism in a sequential manner. This sequence is called the “Genetic Code”. Each gene occurs in two characteristic forms, each called an “allele”. For example in humans, there can be one allele for brown hair and one allele for black hair. Most of the time one allele is dominant and other is recessive. The combination of allele determines the characteristics of an individual. The observed characteristic of an individual is called the *phenotype* and the genetic code of the character is the *genotype*.

The building blocks of genes are Deoxyribonucleic Acid (DNA) and it exists in a double helix shape. DNA looks like a long twisted ladder when seen under the electron microscope. DNA consists of two strands of sugar-phosphate bases which are the backbone of the helical structure and these are bonded together with four different chemical bases: Adenine (A), Guanine (G), Thymine (T) and Cytosine (C). These bases uniquely combine together in different combinations to form the genetic code. The DNA string is very thin (2nm) and it is packed into a structure called a chromosome. The whole package is housed in the nuclei of an organisms’ cells. The picture shown in figure 3.1 gives an overall view of a cell, chromosome, DNA and genes.

Thus, parents’ traits are represented in the chromosomes of each cell. These traits are passed to offspring by the process called “Reproduction”. This is explained below.

Single celled organisms reproduce by simple cell division, which is known as *mitosis*. In this case, the chromosome is exactly copied and hence the offspring will be identical to the parent. Higher organisms reproduce by sexual reproduction, known as *meiosis*. In both cases, during cell division, errors occasionally occur when copying genes from the parents. The cause of these errors can be internal or external.

Internal errors occur during the process of copying the chemical bases. In humans, approximately six billion of these chemical bases (As, Ts, Gs and Cs) have to be copied and some errors are inevitable. External errors are those caused by such things as ultraviolet light and X-rays [3]. The changes in genes affect the traits passed to the offspring. These changes are called "*Mutations*". Mutations cause changes in the genetic code at random points.

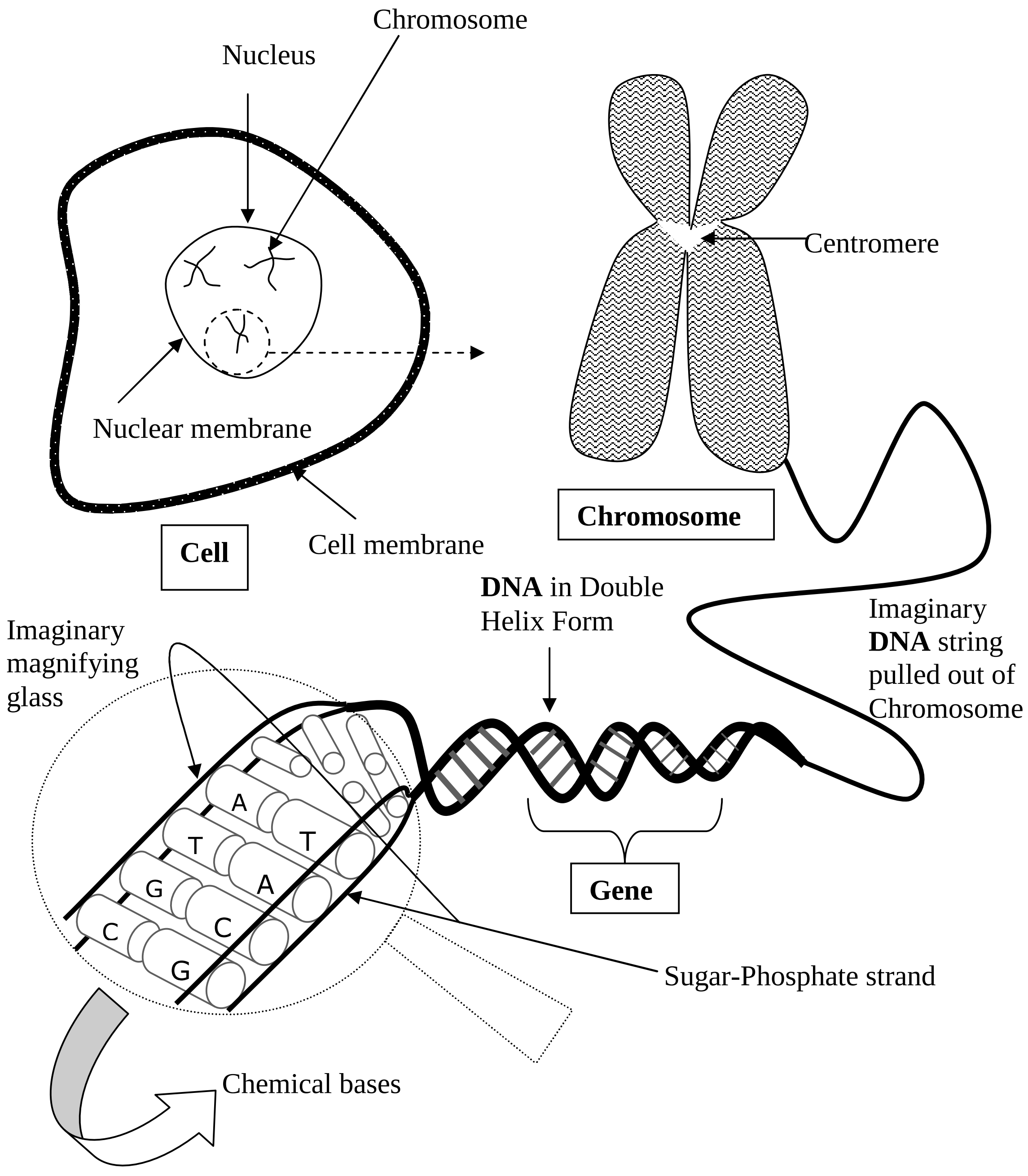


Figure 3.1 Cell, Chromosome and DNA

During sexual reproduction, the offspring gets on average 50% of the male chromosome and 50% of the female chromosome. These chromosomes combine together at random points. This is called "*Crossover*". Thus traits are passed from generation to generation.

So far the basic components of inheritance have been discussed. The description below shows how the basic components of inheritance influence the evolution of the organism itself. Many researchers and philosophers have contributed to the ideas behind evolution; but it was Charles Darwin, born in 1809, who came up with the theory of "Natural Selection". He initially went to Edinburgh University to study medicine, but it did not interest him. He left Edinburgh and joined Cambridge University to train for the church; however, he did not enjoy that either. His uncle persuaded his family to let him join a journey around the world on a ship called "Beagle", hoping that, being at sea would teach him about life and how tough it was. In 1831 he began his journey as a naturalist aboard the Beagle [4]. On his journey, between 1831 and 1836, many things in the natural world inspired him to think about the evolution of living organisms. When the journey had finished, he started theorising about the relationship between organisms and the environment. After observing many different species, he came up with a theory called "Natural Selection". This theory is laid out in a point by point fashion [5] as follows:

1. *The prime motives for all species are to reproduce and survive, passing on the genetic information of the species from generation to generation. When species do this, they tend to produce more offspring than the environment can support.*
2. *The lack of resources to nourish these individuals places pressure on the size of the species' population. This lack of resources means increased competition and as a consequence, some organisms will not survive.*
3. *The organisms who die as a consequence of this competition are not totally random. Those organisms more suited to their environment are more likely to survive.*
4. *This results in the well known phrase, "survival of the fittest", where the organisms most suited to their environment have more chance of survival if the species falls upon hard times.*

5. *Those organisms who are better suited to their environment exhibit desirable characteristics, which is a consequence of their genome being more suitable to begin with.*

Darwin described "Natural selection" as the process of selecting the fittest individuals for further reproduction. An example, from Darwin's observations when he visited the Galapagos Island will help to illustrate this better. On one island there were small tortoises which eat the lower leaves of the trees to survive. During the dry season, the tree's lower leaves die. In such situations, the small tortoises die from starvation. But large tortoises, which are able to reach the height of those leaves left, eat well and survive. See figure 3.2.

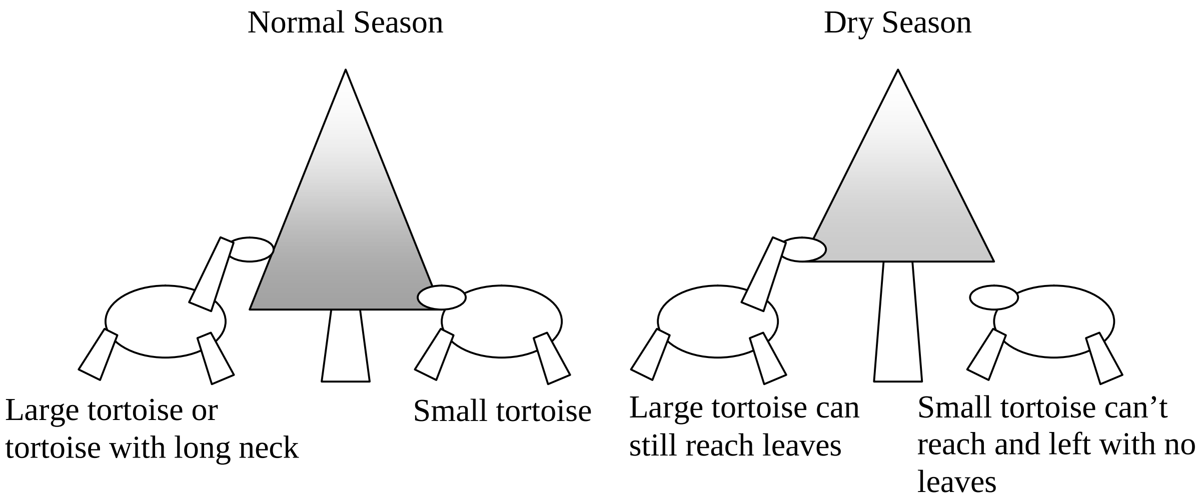

Normal Season
Dry Season
Large tortoise or tortoise with long neck
Small tortoise
Large tortoise can still reach leaves
Small tortoise can't reach and left with no leaves

Figure 3.2 Example of "survival of the fittest"

If the dry season continues the small tortoises will die out. The large tortoises would survive and therefore would be able to pass their genes to the next generation.

To summarize the main points from above:

1. A group of individuals is called a population
2. The individuals within the population carry their genetic information in their chromosomes
3. The genetic information is coded in the chromosomes as genes
4. The genes themselves are made from DNA
5. In sexual reproduction chromosomes crossover to produce new offspring

6. Mutation produces new codes and happens randomly
7. According to the ability to survive, the fittest offspring survive and the others die
8. The surviving individuals pass their genes to next generation

Thus the species gets biologically optimized to fit in with the dynamic environment. This concept is the basis for most of the EAs as the following sections explain.

3.3 The Genetic Algorithm – an overview

The GA was developed by John Holland in 1975 and popularised by one of his students, David Goldberg in 1989. Goldberg in his book defined the GA as follows [6]:

"Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search".

The "Tortoise on the Island" example, above, can be used to explain how the GA works. Imagine a mixture of interbreeding tortoises in a population. The features of each tortoise can be any two of the following: large, small, hard shell and black. The ultimate aim of tortoises is to survive. The environment the tortoises are living in is dry and hence it is hard for them to reach the lower leaves of the trees. The island also has predators. Figure 3.3 gives a pictorial view of the process of the GA. In the figure, the numbers on the tortoises represent their features. In the GA's process these numbers are taken as genes representing the system.

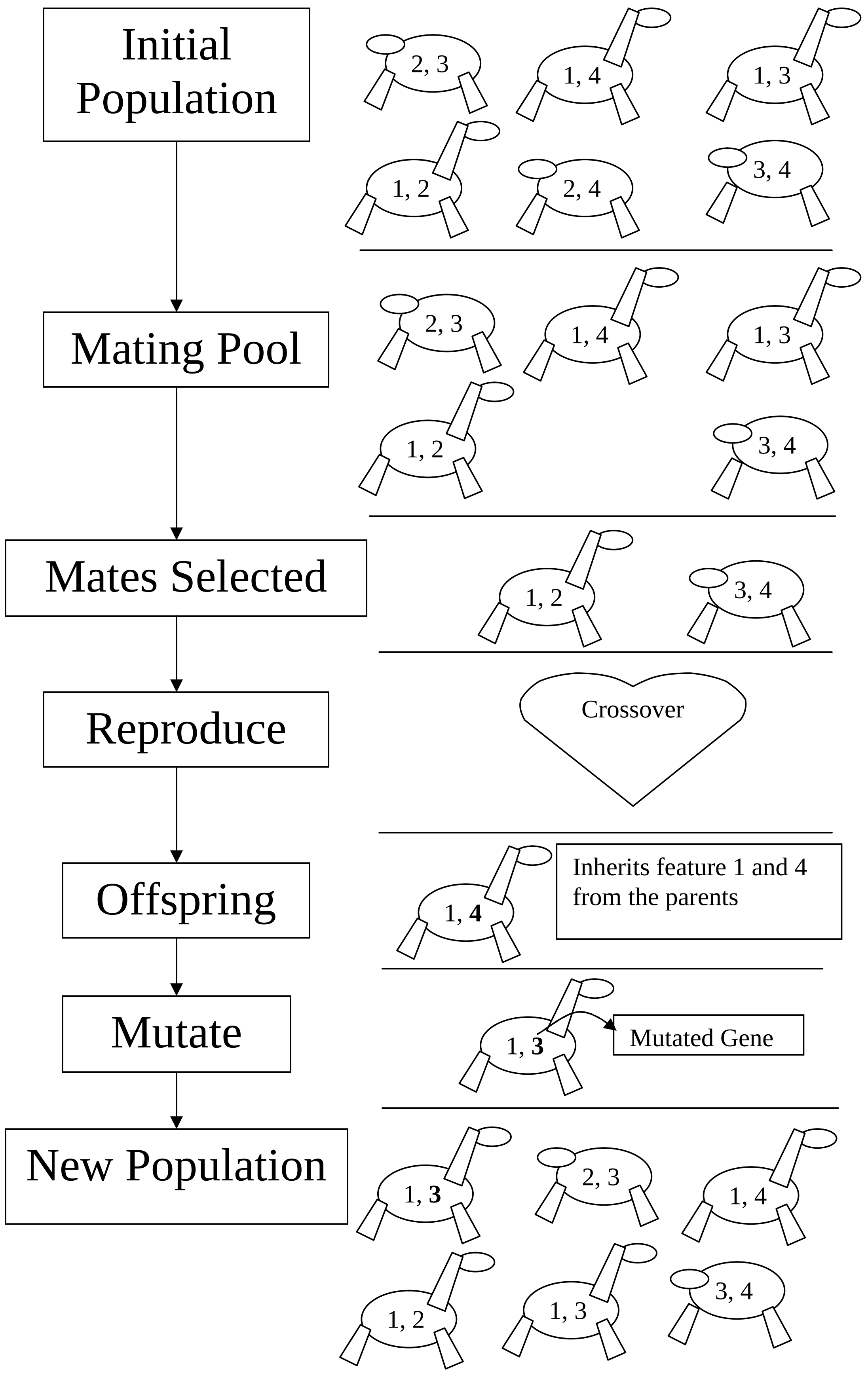


Figure 3.3: Process of the GA

1 – Large **2** – Small

3 - Hard Back **4** - Black colour

From the initial population, not all tortoises make it into the mating pool. Only those which are fit enough are chosen by the GA. From the mating pool, two are randomly selected for reproduction. During reproduction, each offspring inherits genes from its parents which decide its features. Then, randomly, some genes go through the process of mutation where their features may be varied. This process is explained in detail later in this chapter. Mutation may result in a good or a bad change to the off-spring's feature. In the example below, mutation resulted in a good change, because it got a hard back, a feature which allows escape from predators. After one iteration, the new population has two tortoises which are fitter the than others (genes 1, 3). After several generations or iterations, others will become extinct and thus the population gets fitter. As the GA works on the genes representing the system, it is known as *genotypic* algorithm, an analogy to the *genotype* in biology (refer to section 3.2: Biological Optimization).

In general, the GA tries to optimize a system's parameters to maximize the fitness (or minimize the error or cost) of the system, in each iteration.

Generally, the point where the GA finds the true minimum error is called the *Global Minimum*. If it fails, it is referred to as having found only a *Local Minimum*. The difference between local and global minimum is illustrated in figure 3.4.

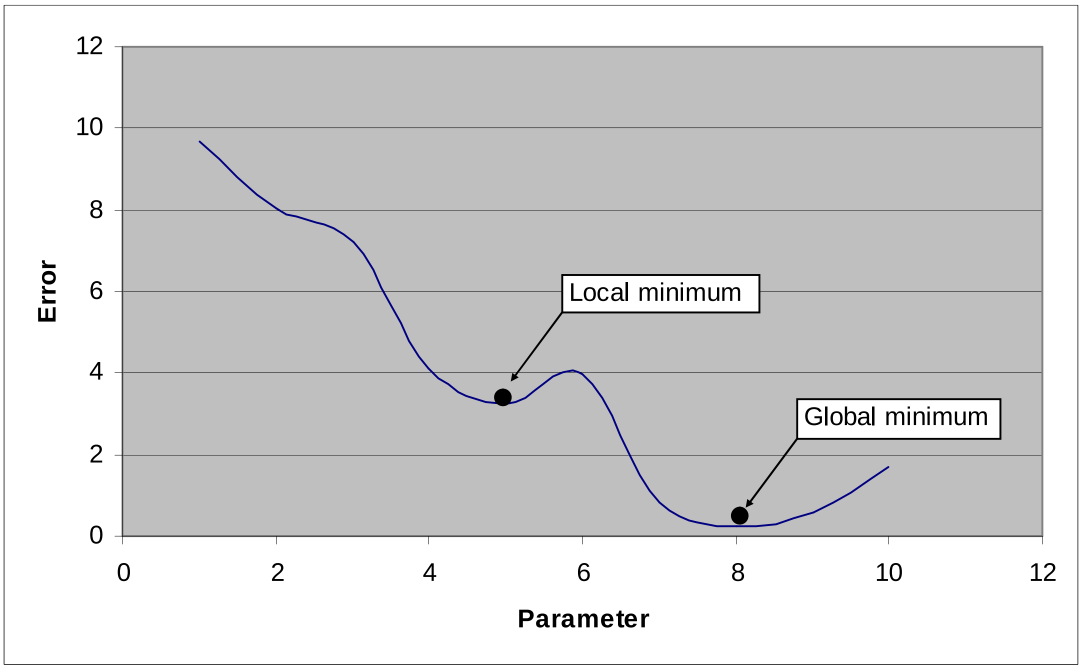


Figure 3.4 Local and Global minimum

3.3.1 Types of GAs

There are two main types of general GA: “The Binary GA” and “The Continuous Parameter GA”. These two types differ in terms of how the parameters of a problem are represented. In a binary GA, for example, the parameters x = 4 and y = 6 would be represented as binary values x = [1 0 0] and y = [1 1 0]. However, in many problems, parameters are taken as they are represented in the system. These parameters often belong to the continuous type. Continuous parameters are real numbers, for example 1.253, -0.836, etc. Due to the close similarities between the binary GA and continuous parameter GA, in this thesis the binary GA is not considered. The next section explains the continuous parameter GA.

3.3.2 The Process of Continuous Parameter Genetic Algorithm

The different processes of the GA are shown below in a flow chart (Figure 3.5).

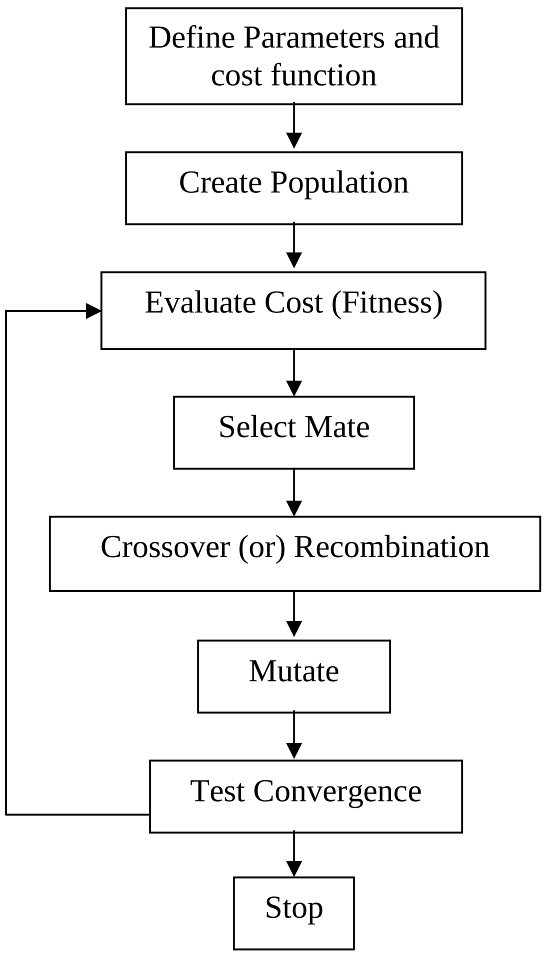


Figure 3.5 Block diagram of the process of the GA

Each processing stage is explained in detail in the sections below.

3.3.2.1 Defining the Cost Function

The main aim of using a GA is to optimize system parameters - to reduce the error of the system. To do this, the GA needs a function which evaluates the system parameters. This is generally called the "Cost Function" or "Fitness Function". The Cost Function can be defined as the process that uses the input parameters from a given problem and evaluates their cost (or fitness), according to the constraints of the given problem. Figure 3.6 shows a block diagram representation of the cost function and problem constraints in the GA. For example, in a problem to solve the equation (3.1),

$$x = 2\sin(p) + 10\cos(q) + 11r + s/1000000 \qquad (3.1)$$

with condition $0 \leq x \leq 10$

The cost function evaluates the cost of the input parameters 'p', 'q', 'r' and 's', satisfying the condition $0 \leq x \leq 10$. Hence, here the cost of 'p', 'q', 'r' and 's' is being evaluated with 'x' being within the given bounds.

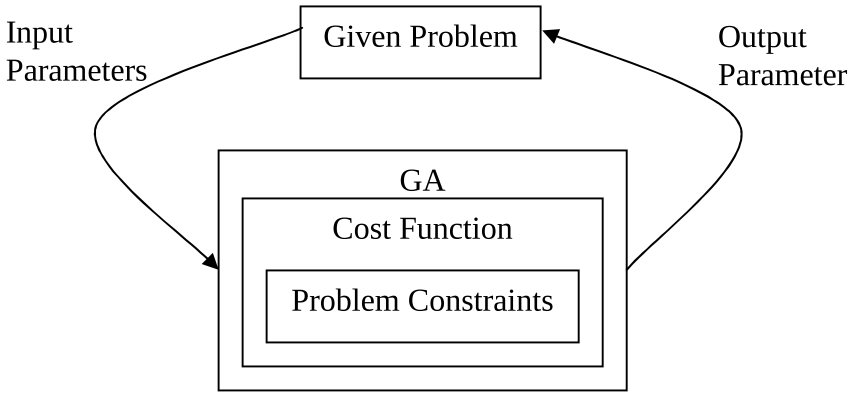


Figure 3.6 Block diagram of the cost function and the problem constraints

The cost function can be mathematical, the outcomes of a game, the performance of an operation, etc. Sometimes, designing a cost function for a given problem is a matter of trail and error. In the example above, the cost would be how close 'x' is to the desired value (the closer it is, the lower the cost).

3.3.2.2 Defining the Parameters

Usually, the GA starts by defining an array of parameters which is called a chromosome or string. From the above example, (equation 3.1) the array of parameters would be the values of ‘p’, ‘q’, ‘r’ and ‘s’. An example of a chromosome in the above problem is shown in figure 3.7, below.

p	q	r	s	
12.2	0.234	-0.534	-2.344	Chromosome

Figure 3.7 A sample Array of parameters – a Chromosome or a String.

The values ‘p’, ‘q’, ‘r’ and ‘s’ determine the characteristics of the equation. This is analogous to the biological genes determining the characteristics of a species. Hence, sometimes researchers use the term “genes” to mean problem parameters. When the cost function measures the cost of the chromosome above, it evaluates the equation as shown in equation 3.2.

$$x = 2\sin(\mathbf{12.2}) + 10\cos(\mathbf{0.234}) + 11 \text{ x } \mathbf{-0.534} + \mathbf{-2.344}/1000000 \qquad (3.2)$$

There are a few points to be considered when choosing the parameters to be used.

1. **Choosing the right number of parameters:**

   Many systems have a lot of parameters which could be chosen. Choosing too many parameters can slow down the GA; it can therefore be quite tricky to choose a suitable number. Sometimes, a trial and error method is used and sometimes the nature of the problem helps to indicate the correct parameters. For example, from the above equation 3.1, the parameter ‘s’ is divided by one million, so it has little effect on the output. Hence, that parameter can be excluded from the evaluation of the cost function and the equation is simplified to:

   $$x = 2\sin(p) + 10\cos(q) + 11r \qquad (3.3)$$

   Thus the number of parameters is reduced from four to three (p, q and r) and the search space also reduces.

2. **Choosing the right types of parameters**

   There are generally three types of parameters one can choose from.

   a. Constrained Parameters

   Sometimes optimization problems require parameter constraints to reduce the workload on the GA and thereby reduce the time taken to optimize parameters. Constrained Parameters come in three types.

   i. Parameters with a hard limit

   Parameters can be imposed with a hard limit such as < (less than), > (greater than), ≤ (less than or equal to) and ≥ (greater than or equal to). For example: in the above problem, a constraint of only positive values for parameter ‘p’ ($p \geq 0$) may be imposed because cos(20) = cos(-20) and hence, there is no need to consider negative values for parameter ‘p’. Thus, hard bounds help to reduce the time taken to optimize some parameters.

   ii. Transform Parameters

   In this case, changing one value of a parameter within its bound will transform another value of a parameter out of its bound. For example, in equation $x = \sin y + 2.2$; $0 < x < 10$, changing the ‘x’ value within its bound transforms the ‘y’ parameter outside its bound and visa versa. Because of this, many unconstrained problems become constrained.

   iii. Finite Parameters

   Choosing finite parameters is choosing from a discrete set of values within the region of the solution. For example, if there is a problem to choose the right number of chairs to be placed in a hall, the GA has to pick the numbers from integer numbers like 1, 2, 3,...etc. and not real numbers like 1.3, 2.3,...etc.

b. *Dependent Parameters*

   Dependent parameters are ones where changing one value will affect others. Haupt, in his book, has given a good example of a dependent parameter [7]. For a car, size and weight are dependent parameters. When the size of the car is increased, it is more likely to increase the weight, unless some light-weight material is used.

c. *Independent Parameters*

   Independent parameters are those which will not interact with each other. With this type, adding extra parameters will not affect the pre-existing parameters. Again, Haupt's book has a good example [7]. In Fourier series, if ten coefficients are not enough to represent the function then more coefficients can be added without changing or recalculating the original ten coefficients.

As seen from the above, there are different ways of classifying a problem's parameters and the GA is sensitive to each of them. By carefully defining the parameters, one can improve the performance by the GA.

Summary of main points of this section

- The constraints of a given problem are an important factor in evaluating the cost function. Sometimes the cost function can be designed by a trial and error method.
- Defining parameters carefully, improves the performance of the GA.
- Sometimes, the choosing of the parameters and the cost function are interrelated.

3.3.2.3 The Initial Population

The initial population is a set of random numbers generated for the parameters of a given problem. Each such set is called a chromosome or string (as explained in section 3.3.2.2). Most GAs are programmed using software (such as MATLAB, C++, etc) although they can be implemented in hardware. Creating an initial population is generally quite easy using such software. Often, the method is to assign a two dimensional matrix. This matrix is an array of chromosomes. In the array, each row represents a set of parameter values (a chromosome) and the whole matrix represents the initial population. For example, for the problem given in section 3.3.2.2, (equation 3.2) a typical array of chromosomes is shown in the table in figure 3.8.

Initial Population			
p	**q**	**r**	**Chromosome**
0.359146	0.368715	0.588958	**1**
0.968575	0.209526	0.700539	**2**
0.623204	0.873273	0.079489	**3**
0.565509	0.191972	0.595049	**4**
0.661988	0.610982	0.376377	**5**
0.185029	0.773646	0.180691	**6**
0.055999	0.834726	0.586256	**7**
0.092188	0.932479	0.984087	**8**
0.580009	0.442063	0.483987	**9**
0.715647	0.446312	0.107349	**10**

Figure 3.8 An array of chromosomes

In an initial population, each randomly generated chromosome can be a potential solution in search space (search space being the total area of potential solutions). For example, the optimized parameter values for the equation 3.3 are between zero and one. A pictorial view of the sample search space and the random potential solution points (the initial population) is shown in figure 3.9. The points shown in the graph are the randomly selected potential solutions.

Although the initial population may be randomly scattered in search-space, as suggested above; the user may also choose to distribute it evenly (or a mixture of the two).

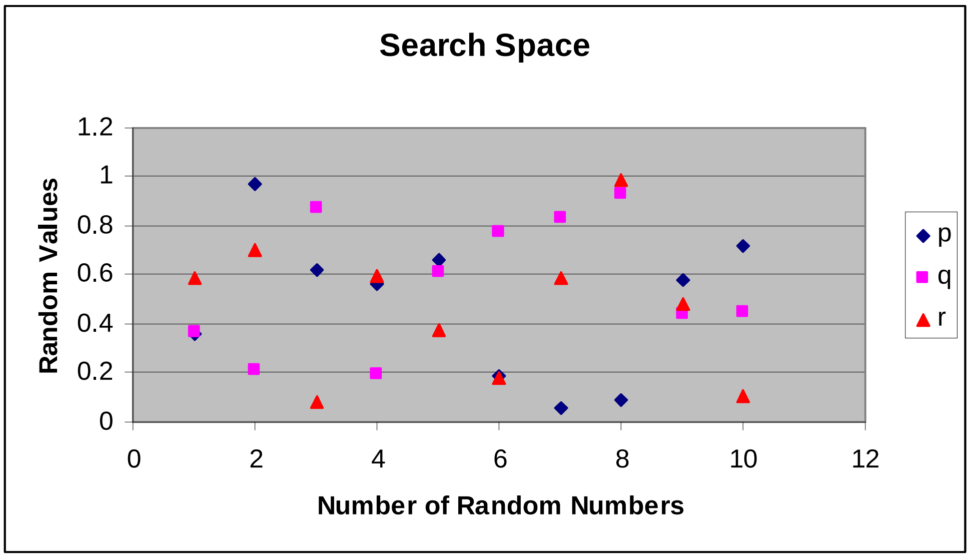


Figure 3.9 Search Space

3.3.2.4 Selecting Members of Population

After generating the initial population, the next step is to select the fit members or chromosomes. A simple method of selection has the chromosomes sorted according to their fitness or rank - from the most fit (first rank) chromosome, to least fit (last rank). Often, from the initial population, only the top 50% is taken for further processing and the bottom 50% is deleted. This top 50% is generally called the breeding population. The size of this population is often then kept constant throughout the rest of the process. As an example of this process: if the initial population size is 48, after sorting according to their fitness or rank, the top 24 fit chromosomes are selected.

3.3.2.5 Selection Operators

This is the process of making pairs from the parents selected from a population. The main idea behind selecting parents is to give more chance to the fit members of a population to reproduce. This closely mimics the survival of fittest as explained in section 3.2. There are many different methods of doing it [9]. The common methods used are “Roulette Wheel” and “Tournament Selection”. These two methods are explained below.

1. **Roulette wheel**

   The Roulette Wheel Method randomly picks parents from a roulette wheel which is portioned in proportion to the cost of the parents [10]. Figure 3.10

shows an example of a roulette wheel portioned in such a manner with the cost of five parents.

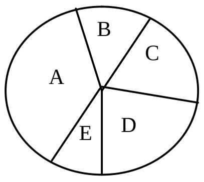


Figure 3.10 A sample roulette wheel

In this example, the chromosomes are represented using the letters A, B, C, D and E. Hence, when a ball is spun on the wheel, it falls randomly into one of the slots and picks a chromosome. In this method, the fittest chromosomes are more likely to dominate the pairing process because they have a large portion of the wheel.

2. **Tournament Selection**

   This method randomly picks two parents from a subset of the mating pool [11]. Among the two the one with the lowest cost is selected for mating. This is useful when the GA works with large populations, because its saves time sorting the cost or rank. The table below (figure 3.11) shows an example.

Tournament	Two randomly selected parents	Winning parent
1	1, 4	1
2	4,6	4
3	3,4	3
4	5,6	5
5	1,6	1
6	2,3	2

Figure 3.11 Tournament selection

There is no definitive guide to the best selection method. Different researchers use different techniques.

3.3.2.6 Mating

Mating, also known as Crossover or Recombination, is the process where the parents combine their genetic material and produce one or more offspring. Generally, in a pair of chromosomes, one is called the father and the other one is called the mother. The main idea behind the mating process is to pass the parent’s traits to the off-spring. It is usually done by taking a portion of genes from the father and a portion of genes from the mother and combining them together to produce off-spring. The point where portions from mother and father are split is called the crossover point and is chosen at random. Figure 3.12 shows an example.

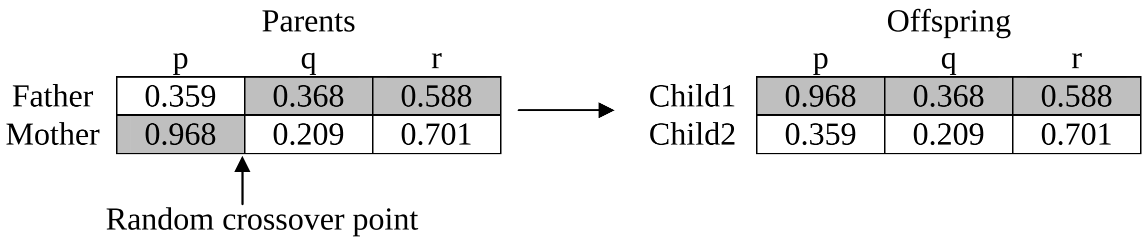


Figure 3.12 Crossover

From the above figure, it can be seen that the portions highlighted belonging to the parents, combine together to form child one and two respectively. Sometimes, instead of picking one random crossover point as shown above, more than one point is selected to produce more offspring. There are also crossover methods developed for specific applications [12]. Although the above methods produce new offspring, parental characteristics (genes or values) remain the same. Hence, the values of the initial population remain the same throughout the process and the GA has to rely on the mutation operator (mutation is explained in the next section) to generate new genetic information (new values). Haupt in his book [13], has outlined work done by Radcliff, Michalewicz, Eshelman and Shaffer, which has introduced new methods to overcome this problem. Their work is outlined below.

Radcliff introduced a new formula to generate values in the offspring’s chromosomes. The formula is shown below (equation 3.3).

$$Pon = \beta Pmn + (1-\beta)Pfn \qquad (3.3)$$

where: β is a random number generated between 0 and 1

P_{on} is the nth gene in the offspring's chromosome

P_{mn} is the nth gene in the mother's chromosome

P_{fn} is the nth gene in the father's chromosome

Substituting β with (1- β) will give the next offspring. If β = 1, P_{fn} dies and if β = 0, P_{mn} dies. β = 0.5 (Davis, 1991) gives an average of father and mother chromosome values. This method introduces new values into the offspring chromosome; however, the values introduced are within the bounds of the parent chromosomes. To solve this, Michalewicz proposed another crossover method, where the offspring value is determined by equation 3.4, given below.

$$Pon = \beta(Pmn - Pfn) + Pfn \qquad (3.4)$$

This method allows changes in the values of the offspring chromosomes, outside the bounds of the parent chromosomes. The blend (BLX-α) crossover method, introduced by Eshelman and Shaffer, is an alternative technique, which determines how far the new values generated can go beyond the bounds of their parents. This method allows the user to keep the offspring values close to the parent chromosome values. There are also other methods available for crossover.

Like selection operators, the best method is hard to advise. Generally, it is at the user's discretion to choose which crossover method to use for a given problem.

3.3.2.7 Mutation

Mutation is the process by which the parameter values (genes) are altered by small random values. This allows the GA to explore the cost surface further without converging to a solution too quickly and helps to avoid local minima. The number of parameters or genes in a chromosome to be mutated, is called mutation range. The usual mutation range can be anywhere between 1% and 20%, depending on the problem, for the GA to work well. Mutation values are generated using a "Uniform Distribution" of random numbers. The bounds of the random number distribution can either depend on the constraints of the given problem or can be completely random. The graph below shows an example of uniform distributed random numbers (figure 3.13). The x-axis represents the bounds of the random number and the y-axis represents the probability of occurrence. The grey area shows the probability distribution. In other words, all numbers within the range are equally likely to occur.

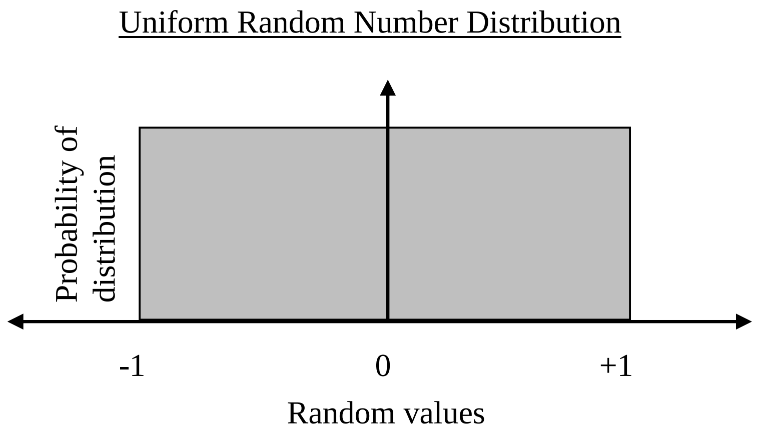


Figure 3.13 Uniform Distribution of random numbers

3.3.3 Advantages of GAs

The main advantage of the GA is that it can efficiently and simply explore a wide range of cost surfaces. Also, it does not result in a single solution, but in several, leaving an option for the user to investigate the system further.

3.4 Simulated Annealing

The Simulated Annealing (SA) method was introduced by Kirkpatrick and his co-workers in 1983. It is based on an idea formulated by Metropolis *et al* in 1953 [14]. Davis, in his book, has defined SA as quoted below [14]:

> *"Simulated Annealing is a stochastic computational technique derived from statistical mechanics for finding near globally-minimum-cost solutions to large optimization problems."*

Kirkpatrick *et al* demonstrated the use of SA when applied to combinatorial optimization problems. Some of the popular problems solved using SA were the physical design of computers, wire-routing, component placement in VLSI design and the travelling salesman problem [15]. In the following section, statistical mechanics is discussed so that SA may be understood in better detail.

3.4.1 Statistical Mechanics

Statistical mechanics is a method used to analyse the behaviour of large systems of atoms in a substance in thermal equilibrium at a finite temperature.

A cubic centimetre of fluid contains in the order of 10^{23} atoms. It is obviously difficult to analyse the behaviour of each atom. Instead, the most probable behaviour of the system at a given temperature is measured. Each configuration of the system may be predicted using the Boltzmann Probability Factor as shown below

$$B_{PF} = e^{-E/k_B T} \qquad (3.5)$$

Where B_{PF} is the Boltzmann probability factor
E is the energy of the configuration
k_B is Boltzmann's constant
T is the temperature

If T is lowered below a certain temperature, the system may form a crystalline solid. Analysing the system in this ground state is difficult because the energy is low. In order to analyse such states, one has to slowly cool the material to its ground state, often starting from the melting point.

3.4.2 Optimisation Techniques from Statistical Mechanics

In optimisation, the analogy to finding the low temperature state of a material is finding the global minimum value. Unlike optimisation using heuristic methods, which often finds local minima, Statistical Physics (SP) provides a better technique to find the global minima. Having said that the technique finds the global minimum, it is not always the case. The reason is as follows.

Imagine if the SP technique is applied to an optimisation problem. The cost function tries to achieve the lowest cost, which is similar to SP finding the low energy state configuration. In a similar way to an iterative improvement method SP accepts only re-arrangements which give a low cost. This, in terms of SP, is analogous to rapidly quenching from a high to a zero temperature, leaving the final solution in a local optimum.

Metropolis et al introduced a simple algorithm which overcomes this problem. It simulates an ensemble of atoms at a given temperature. In each step of the simulation, an atom is displaced by a small random value. Then the energy (ΔE) of the system is calculated. If $\Delta E \leq 0$, the re-arrangement is accepted; if not, then the Boltzmann probability factor is used to decide whether to accept the change or not. The formula is shown in equation 3.6.

$$P(\Delta E) = e^{-\Delta E / k_B T} \qquad (3.6)$$

In practice, this operates as follows. A random number between one and zero is generated using a uniform random number distribution. Let us say that the random number is 'r'. If P(ΔE) > r then the change is accepted and, if not, the previous arrangement is used. The temperature T is the control parameter which is used in the annealing process. Thus, the algorithm prevents quenching quickly and this helps to find the global optimum. This extended technique is the basis of the SA.

3.4.3 Optimisation by Simulated Annealing

The SA, when applied to an optimisation problem, initially generates a random configuration for the system (this stage can be considered as the melting point of physical material). Then the temperature is reduced slowly and random changes are made to the system using the Metropolis method described above. At each temperature, one has to make sure that the system is in a steady state and hence the re-configuration is repeated. The number of times the SA is re-configured at each temperature is known as the annealing schedule.

The random numbers used for changing the values are either generated using a uniform random distribution, as seen above (figure 3.13), or a Normal (also called Gaussian) Random Distribution. A Normal distribution can be defined as [16]:

A Normal distribution in a variate X, with mean (ξ) and variance (σ^2) is a statistical distribution with a probability function

$$P(X) = \frac{1}{\sigma\sqrt{2\Pi}} e^{-(x-\xi)/2\sigma^2} \qquad (3.7)$$

on the domain $-\infty \leq X \leq \infty$.

The Normal Distribution is shown graphically in the figure 3.14. The first graph shows a normal distribution with mean $\xi = 0$ and variance σ^2 (σ^2 is the square of standard deviation) or standard deviation $\sigma = 1$. The second is with mean $\xi = 0$ and standard deviation $\sigma = 1.5$. The graphs show that the probability of small random numbers occurring is higher than for large random numbers: For example, there is more chance of generating random numbers between interval [-1, 1] than between (say) the interval of [-2.5, -1] and [1, 2.5]. Large variance increases the probability of large numbers appearing and small variance increases the probability of small numbers. Hence, the idea of generating big and small random numbers using a normal distribution helps the SA to control the mutation size when the error is being reduced, close to the global minimum. This is known as a controlled mutation step size.

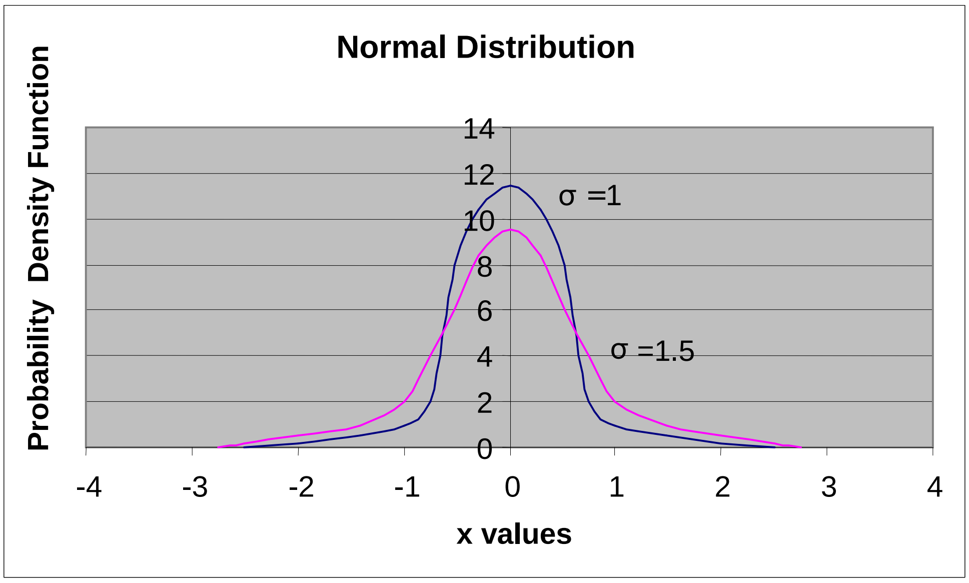


Figure 3.14 Normal distribution graph

3.5 Comparison of the SA and the GA

So far the two popular algorithms, the SA and the GA, have been explained. When these two algorithms' efficiencies are compared in general, it is hard to say which one is more efficient. This is because their efficiency depends on the nature of the optimisation problem to which they are being applied. Some researchers, in a range of different areas, have compared the efficiency of SA and GA [17][18][19] and Jukka Kohonen, in a web site [20] has outlined the theoretical and empirical difference between SA and GA.

As shown above, the SA has only one string of parameters and only uses the mutation operator to get an optimal solution. In contrast, the GA has a population of trial solutions and combines two or more good solutions to get a better one. This difference between the algorithms does not necessarily make one algorithm superior to the other, because both follow the assumption that good solutions are always found close to the already known good solution. Without this principle, the algorithms would not perform any better than a random search. The GA becomes a better choice when the problem's parameters do not affect the resulting solution during recombination. An example of a case where a GA performs badly is a VLSI wire-routing problem. The chromosome would contain the routes of each wire. During recombination, when two such strings are combined, there is a good chance that the resulting string would contain repeats of the same routes. On the other hand, the SA usually does not have such problems because it always makes a change to an existing solution and hence there is little or no chance of getting parameters repeated. Having said that, the SA is usually slow compared to the GA. The SA makes small local moves in the search space, whereas the GA moves fast by combining two good trial solution points in the search space. The SA performs better if the evaluating solutions are always near the existing efficient solution and hence the resulting optimum solution could be very close to the real solution.

3.6 Evolutionary Strategies

Evolutionary strategies (ES) are the result of a joint effort by Rechenberg and Schwefel in the 1960s at the Technical University of Berlin in Germany [21] [22]. The original ES was developed in 1964, to solve a pipe bending problem in manufacturing, as analytical methods had failed to solve it. The team proposed the pseudo-random method which has become known as the Evolutionary Strategy (ES). The result was a great success.

In general the ES algorithm proceeds as follows:

1. It starts with a random potential solution or with a population of random potential solutions (some ESs have only one string of parameters).
2. For each one of them, it produces an offspring using its mutation operator (some ESs also use the recombination operator).
3. For all the parents and the resultant offspring, it calculates their fitness.

4. If the offspring performs better than the parent, then the parent is replaced by the offspring. If not, then off-spring is rejected and the original parent is kept.
5. The process is repeated until the error is reduced.

Over the years, many variations on this theme have been developed and the different ESs are explained later. In some variants of ES, mutation is the main operator, whereas in the GA it is always secondary to crossover. In ES, all the genes are usually mutated in each iteration. Mutation uses a normal distribution with an expectation rate of zero. This means that on an average, it can be expected that there is zero difference between the existing state of the system and the newly mutated system. This is in addition to the advantages previous described in the section on SA.

To generate normally distributed random numbers with a zero expectation value, Schwefel [21] gave a simple formulae (equation 3.8 and 3.9).

$$Z_1 = \sqrt{-2\ln(U_1)}\sin(2\pi U_2) \qquad (3.8)$$

$$Z_2 = \sqrt{-2\ln(U_1)}\cos(2\pi U_2) \qquad (3.9)$$

First two uniformly distributed random numbers are generated (U_1 and U_2). Then U_1 and U_2 are applied to the above equations to get two normally distributed random numbers. To overcome the problem of small random numbers producing poor convergence at the beginning of the algorithm the variance of the normal distribution can be varied according to the error. Taking the square root of the error as the multiplying factor of the variance causes big mutations at the beginning and slowly reduces the mutation size as the error gets closer to the solution. There are also other ways to control the size of the mutation. Rechenberg [21] suggested the "**1/5 Success Rule**" to control the mutation size. The rule can be stated as:

> *"The ratio of successful mutations to all mutations should be 1/5. If it is greater than 1/5, increase the variance; if it is less, decrease the variance."*

The rule suggests that there should be one good mutation for every five: meaning, on average, in five iterations or generations there should be one good mutation. If there are more good mutations, the standard deviation should be increased until it satisfies the 1/5 rule. Similarly, if there are more bad mutations, decrease the standard deviation. Rechenberg showed mathematically that this rule produced the optimum convergence rate in his application.

3.6.1 Different Evolutionary Strategies

There are four common Evolutionary Strategies. Each of them uses a different population size.

1. **(1+1) - Evolutionary Strategy**

   The initial ES had only one parent and one offspring produced by mutating the parent [23]. Hence this was called (1+1) ES and used only the mutation operator. If the offspring produced is better than the parent, then the parent is replaced by the offspring.

2. **(μ+1) - Evolutionary Strategy**

   Later Rechenberg proposed the first multi-membered ES and introduced a symbol μ to represent the number of parents [23]. In this type, μ was greater than one ($\mu>1$). One offspring was produced from a group of parents using recombination and mutation operators. Then the offspring replaced the worst parent among the group. For a summary of ES recombination operators, see reference [27].

3. **(μ+λ) - Evolutionary Strategy**

   Later Rechenberg and Schwefel then proposed another type of multi-membered ES and introduced a term λ to represent the number of offspring [23]. In this type, λ offspring were generated from μ parents using recombination and mutation operators. From a population of μ parents and λ offspring, the best μ were selected for the next generation.

4. **(μ,λ) - Evolutionary Strategy**

   They also proposed an ES where λ off-spring were generated from μ parents [23]. In this algorithm the individuals must satisfy the condition $\lambda \geq \mu$. An innovation in the (μ,λ) ES is that the chromosomes have parental information and mutation step-size built in. Thus, the algorithm learns the mutation size on-line. This is called a self-adaptation process.

In summary, the Evolutionary Strategy is powered by the 1/5 success rule, normal distribution variance step-size control, self-adaptation and recombination techniques and has solved many problems which analytical and other traditional methods failed to solve.

3.7 Evolutionary Programming

The Evolutionary Programming (EP) algorithm was originally developed by Lawrence J. Fogel in 1960 [24]. The EP is similar to ES and only the mutation operator is used. The general process of the EP is

1. It starts with a random population of solutions (similar to the GA).
2. Each individual in the population produces an offspring by using a mutation operator.
3. The fitness of each member in the population is evaluated and sorted according to the fitness.
4. The best half is kept and the rest is deleted.
5. The process is repeated until the error is reduced.

The random numbers are generated using either uniform or normal distribution. As can be seen from the description above, the EP does not use recombination and use only a mutation operator. Originally, the EP operated on the population directly, for example changing the weights of a neural network in situ. Later, it was influenced by the GA's parameter string representation. This is an example of how researchers picked good methods from different algorithms and threaded them together, in order to get a better performance overall. The EP has another advantage in that it may have different sizes of parameter string. This is because it only uses the mutation operator. However, like SA, using only mutation makes the algorithm slow to find the optimum solution.

3.8 Hybrid Algorithms

All the algorithms described so far are similar to each other. Several researchers have investigated the use of algorithms developed using a “pick-and-mix” strategy, meaning that the best parts of these algorithms are threaded together (often with other heuristic search methods). A few examples of this type of algorithm can be found in the references [25][26], where the application of Hybrid EAs are discussed. Eiben and Smith in their book have described a general Evolutionary Algorithm [8], as shown in flow chart below (figure 3.15)

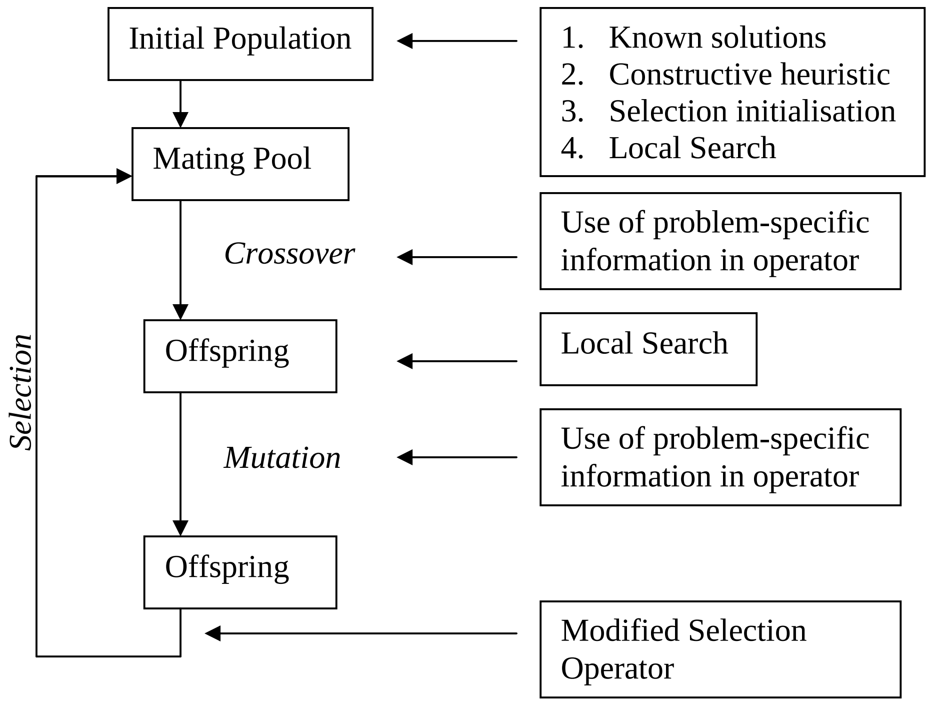

Initial Population
1. Known solutions
2. Constructive heuristic
3. Selection initialisation
4. Local Search
Mating Pool
Crossover
Use of problem-specific information in operator
Offspring
Local Search
Selection
Mutation
Use of problem-specific information in operator
Offspring
Modified Selection Operator

Figure 3.15: Where to hybridise

3.9 Summary

This chapter has given a brief introduction to biological optimization, which is the basis of the different evolutionary algorithms.

In the section on the GA, the two important GA operators, recombination and mutation were described. Other background processes of the GA (parameter representation, initial population, selection process, pairing methods and mating methods) were also discussed.

In the section on the Simulated Annealing, the importance of the mutation operator and its controlled mutation step size, analogous to the cooling process of a hot substance, forming a crystalline lattice, were outlined.

The Evolutionary Strategy section highlighted the mutation operator and its use of normal distribution random numbers and also the different variants of the strategies.

The Evolutionary Programming section explained its use of the mutation operator.

The next chapter will explain how these different algorithms' operators were threaded together to form “The Real-Time Evolutionary Algorithms”, which were used in this project.

Chapter 4

The Real Time Evolutionary Algorithm

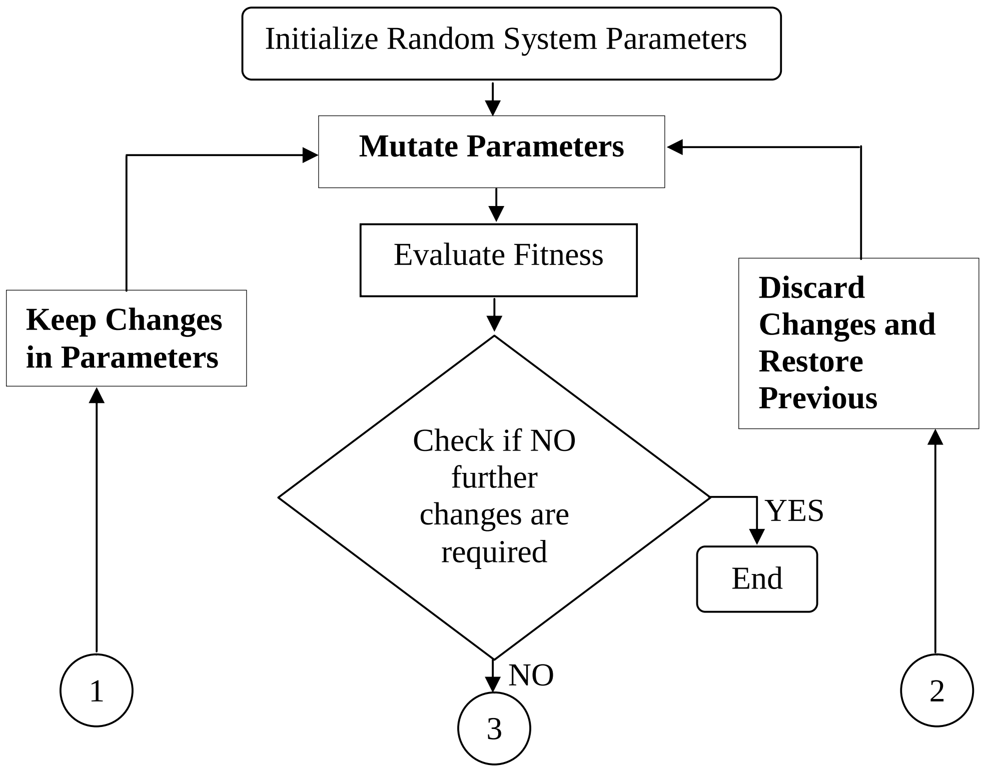
4.1 Introduction

The last chapter covered the important EA operators and their modes of operation. It also explained that new algorithms can be developed using a "pick-and-mix" strategy, according to the required application. A major part of the project was to formulate an effective RTEA. This chapter will outline how the different operators obtained from the principle EAs were evaluated, so that their effectiveness in real-time could be ascertained.

This is a short chapter, which forms the basis of the original research in the project, the rest of the research project being the system implementation, investigation of different neural network topologies and fitness functions which are explained in chapter 6.

4.2 General RTEA

Figure 4.1 below shows the general flow of operation of a RTEA.


Initialize Random System Parameters
Mutate Parameters
Evaluate Fitness
Keep Changes in Parameters
Discard Changes and Restore Previous
Check if NO further changes are required
YES
End
NO
1
2
3

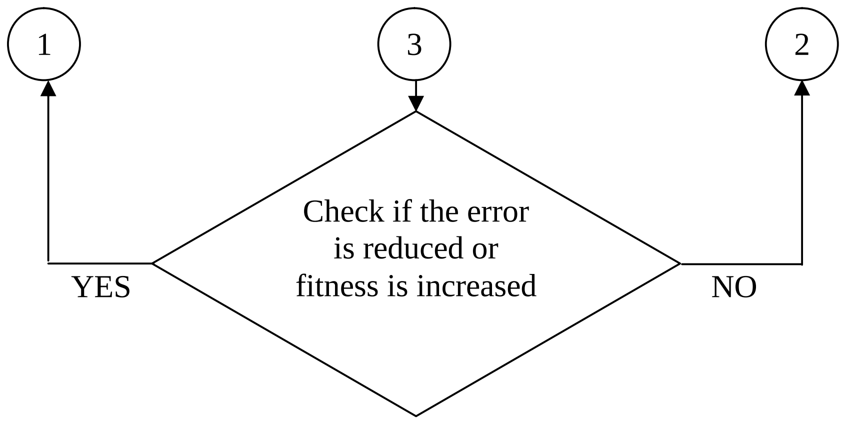


Figure 4.1 General RTEA

In the above figure, the highlighted operators are those which were investigated to establish their effectiveness as explained in the sections below.

4.2.1 Recombination

Figure 4.1 above shows that the RTEAs investigated did not use a recombination operator. The reason for this is as follows. The aim of the RTEAs in this project, as described in chapter 1, is to continuously train a robot's controlling neural networks, on-line. As it is assumed to operate on a single physical robot, in an actual environment there can be only one set of network parameters. Using recombination would require at least two such sets and technically the operation becomes off-line because off-line training simulates the system, evaluates the cost and recombines the strings (as seen in chapter 3). Even if recombination were used on-line (by some complicated means), the system would have to stop or pause its current action in order to complete the cost evaluation and recombination operations. This would obviously slowdown the system and negates the purpose of the RTEA being operated on-line. Hence, the recombination operator is not used. The following sections consider the operators which *are* used.

4.2.2 Mutation Techniques

In the absence of recombination, mutation is the most important operator used to cover search space. This section details the important mutation techniques investigated and also introduces some new terminology used in the investigation. Mutation was applied in three different ways to the system.

1. **Mutate All –** The Mutate All (MA) technique mutates all the degrees of freedom of a system together. This means that the MA changes all the parameters of the system at each step. Accepting the changes made is subject to the conditions of acceptance in the fitness function - which is explained later in the chapter. MA is typical of ES type algorithms, which mutate all of their parameters in each iteration.

2. **Mutate One –** The Mutate One (MO) technique mutates one degree of freedom of a system at a time. The MO chooses one degree of freedom randomly, subject to conditions as explained above. Such a strategy is typical of the mode of operation of a Genetic Algorithm, where mutation is only applied (typically) to single parameters within a string.

3. **Mutate Some –** The Mutate Some (MS) technique mutates selected degrees of freedom of the system together. For example, in a four legged quadruped robot, the MS may choose two degrees of freedom, such as the two left or right side legs of the robot or either of the diagonal legs. The MS strategy chooses the coupled degrees randomly. The reason for trying the MS strategy is that the robot, being symmetrical with bilateral symmetry (and similarly symmetrical gaits), may respond better to symmetrically applied mutations. This technique is intermediate between the MO and MA techniques and is specifically relevant to such symmetrically configured systems.

4.2.3 Random Number Distribution

Each of the mutation techniques described above was tested using both Uniform and Normal Random Number Distributions (see section 3.3.2.7 and 3.4.3).

1. **Uniform Distribution**

   While using a Uniform Random Number Distribution, the effect of varying mutation size was tested. Figure 4.2 below is a sample graph showing how mutation size could vary for a particular problem.

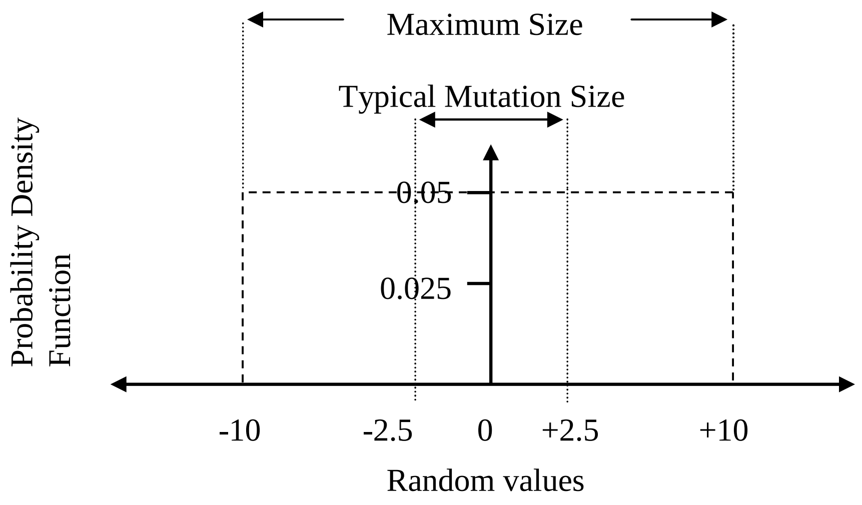


Figure 4.2 Sample graph showing a Uniform Distribution mutation size

The above sample figure 4.2 shows that the maximum range is -10 to +10 and the typical mutation size for a particular problem might be -2.5 to 2.5.

2. **Normal Distribution**

   In a similar way to the Uniform Distribution, the effects of varying the mutation size of the Normal Distribution were tested, as was the effect of increasing and decreasing the variance with error in a similar way to that employed by the SA algorithm. Figure 4.3a shows the mutation size for a particular problem and figure 4.3b shows the increase in variance.

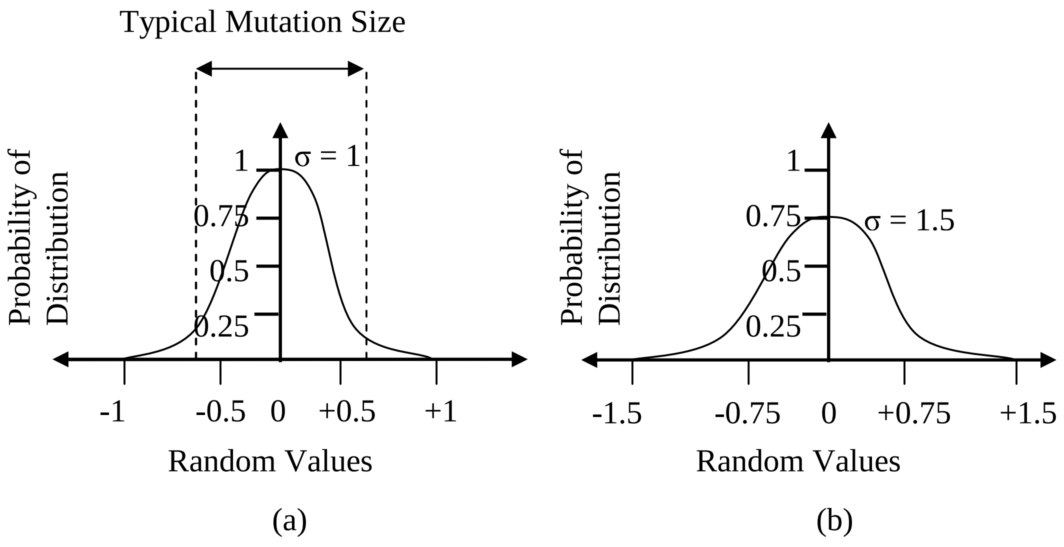


Figure 4.3 Sample graphs showing Normal Distribution mutation sizes

4.2.4 1/5 Rule

Rechenberg's 1/5 rule (as explained in section 3.6) was considered in the investigation, so that its effectiveness in on-line operation could be evaluated. As the rule suggested, if there were more good mutations, the variance was increased (as shown in figure 4.3b) until the system satisfied the 1/5 rule. If there were more bad mutations, the variance was decreased. This should finely tune the parameters of the system when they are close to the optimal state.

4.2.5 Condition of Accepting Changes

After mutation, the fitness of the system was calculated (more detail about the fitness function is given in the next chapter). Depending on the fitness, the mutations were then accepted or rejected. The effects of Simple Acceptance and Occasional Acceptance were tested. This is explained below.

1. **Simple Acceptance**

   This is the simplest algorithm. If the fitness increases (that is, the error decreases), the current mutational change is accepted. If the error increases, the change is rejected and the previous parameters are restored.

2. **Occasional Yes when No**

   A more complex algorithm is that derived from the SA where an occasional increase in error is accepted. As explained in chapter 3, this is a method of avoiding local minima. The probability of accepting an error-increasing change is given by the probability distribution explained in section 3.4.2. The formula used to calculate this probability is shown again below.

$$P(\Delta E) = e^{-\Delta E / k_B T} \qquad (4.2.5.1)$$

All the above techniques were tried out and their performances were compared to one another using the measures described in the following chapters. The results of this investigation allow us to evaluate critically the various operators available. This in turn, allows recommendations to be made as to the components of a suitable RTEA for different circumstances.

4.4 Summary

This chapter outlined the important real-time evolutionary operators that were tried out in this research. It explained that, as the RTEA operates on-line, there is no recombination operator. Chapters 6 and 7 explain the system implementation, the network topology and the initial observations. The next chapter covers the literature of this research.

Chapter 5

Literature Review

5.1 Introduction

This chapter reviews the work by other researchers, which is in the same or a similar area to the research presented in this thesis. The chapter will cover a brief overview of robotics, which includes its history, robot control systems, important work by Hugo de Garis and other robot control systems. It also covers the use of real-time EAs in ANNs and Robotics as well as other methods of control, combining ANNs and Robotics. Finally, a summary will put the research presented here into context with the reviewed work.

5.2 Robotics Overview

5.2.1 History

In 1921, a Czech playwright, Karel Capek, used the word “Robot” in his play “Rossum’s Universal Robots”. He derived the word “Robot” from “Robota”, meaning “forced labour”. Later, in 1942 the American scientist and Science Fiction writer Issac Asimov used the word “Robotics” in his novel, “Run-around” [1]. Since then, actual robots were developed and used in various fields, particularly in industry. The main idea behind robotics is to automate tasks with minimal human intervention. The Robotics Institute of America (RIA) considers a robot as [1]:

> *“A robot is a re-programmable, multi-functional manipulator (or device) designed to move material, parts, tools, or specialised devices through variable programmed motions for the performance of a variety of tasks.”*

A detailed history of robotics is given in a table by Maurice Zeldman in his book [2].

5.2.2. Robot Control Systems

5.2.2.1 Subsumption Architecture

In 1986, Rodney Brooks introduced a control system for mobile robots [3]. The control system consists of a hierarchical layered structure, each layer of which defines a unique behaviour. The level of intelligence increases in higher layers. For example, the lowest layer might be responsible for moving the vehicle without collisions and the immediate next layer for recognizing different objects to assist in movement towards a goal object. Subsumption architectures are usually shown as in figure 5.1.

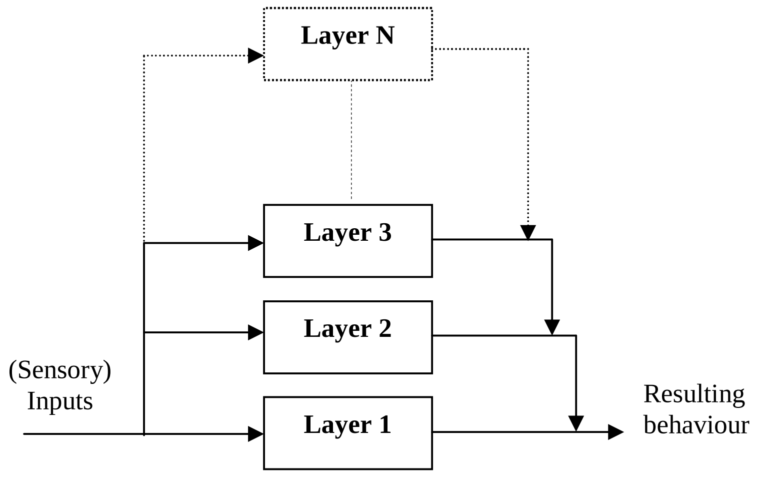


Figure 5.1: A Subsumption Architecture

Figure 5.1 may give the false impression that subsumption architecture is simple. Actually, it is more complicated, because each layer may also contain several modules. The layers are wired through suppressive connections. Each connection is taken from the internal wiring in the higher layer to the internal wiring of the lower layer. This makes the connectivity between layers complex, but helps to simplify the overall design as the same modules do not have to be recreated in every layer. Connell used subsumption architecture to control a robot which picked up disposal cans in offices [4]. Rosenblatt modified this by splitting the modules into smaller decision making units [5]. Maes introduced a behaviour selection mechanism, allowing different behaviours to be selectively activated and suppressed [6].

5.2.2.2 Hugo de Garis

Hugo de Garis introduced a method of Robot Control which evolves an Artificial Nervous System (ANS), using Genetic Programming (GP) [7]. The system was developed using GenNets. GenNets are neural networks which perform a specific time-dependent task. The user can specify their parameters – that is, the parameters of the neural network, evolutionary system and of the desired behaviour (at a low level). Then the GenNets are evolved using a GA. As the GenNets evolve to satisfy all the desired behaviours, they are combined with GP to form the ANS. More of these units can then be evolved and combined using GP, producing a hierarchical nervous system. Using this idea, de Garis generated a walking gait for simulated stick legs. This was used to develop an ANS for an artificial lizard called “LIZZY”. De Garis showed that LIZZY could display a number of different behaviours [8], like approaching prey or mates and fleeing from predators.

De Garis also used GenNets [9] to control the time dependent walking behaviour of a simulated biped using similar stick legs (shown in figure 5.2).

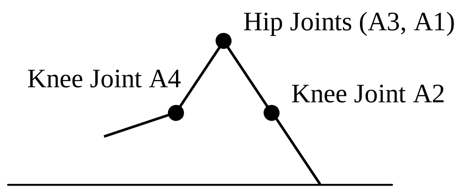


Figure 5.2 Simulated Biped Stick Legs
(After de Garis)

2.5.5.3 Other Robot Control Systems

Generally, other control systems have three main components, as shown in figure 5.3, and are known as Three Layer Architectures (TLA) [10].

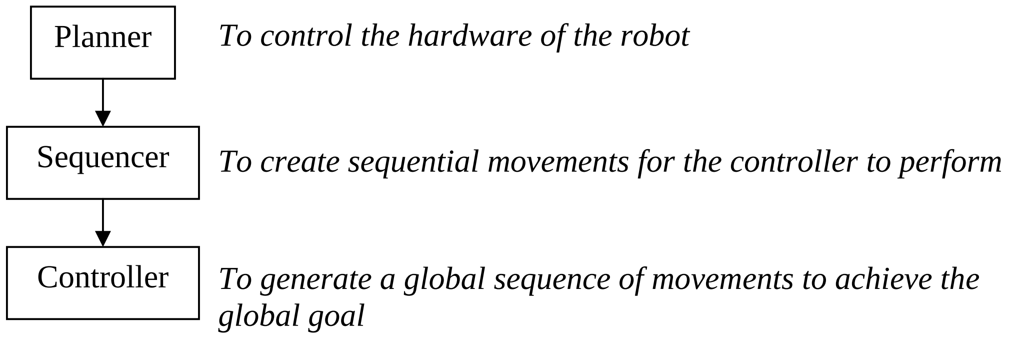

Planner
To control the hardware of the robot
Sequencer
To create sequential movements for the controller to perform
Controller
To generate a global sequence of movements to achieve the global goal

Figure 5.3 General Flow of Control in a TLA

Saridis introduced the principle of increasing precision, decreasing intelligence (IPDI) [11]. In his model, the top planner layer has the most intelligence, but is less precise because it generates broad movement specifications. The precision increases as the layers move from top to bottom, but the intelligence of each layer decreases. The controller has the most precise action as it directly controls the hardware (robot). Examples of similar types of systems may be found in references [12], [13], [14] and [15].

Degney introduced a control structure which is generated as the robot interacts with its environment [16]. The control strategy for particular sensory conditions of the robot is a combination of primitive actions and other control strategies. Such strategies are nested to form a hierarchical control structure.

5.3 Real-Time EAs in ANNs and Robotics

Yang and Meng introduced a neural approach, for real-time motion planning, with obstacle avoidance, of a mobile robot in a non-stationary environment [17]. The neural network, used for collision-free path planning, is a manually set pre-trained network. There is no learning method and therefore no cost function to measure the fitness of the robot, although its motion is controlled by a local selection of the maximum neural activity.

Similar work by Vadakkepat et al. [18], proposed an Evolutionary Artificial Potential Field (EAPF) for real-time optimal path planning, combined with a GA. A Multi-Objective Evolutionary Algorithm (MOEA) was used to select the optimal potential field function (fitness function), where the criteria were goal-factor, obstacle-factor, smoothness-factor and minimum-path-length-factor. The objectives of EAPF were:

1. *To design a simple Artificial Potential Field Function with tuneable parameters, for real time application.*
2. *To derive the associated cost functions.*
3. *To optimize the parameters of the Potential Field Function with the MOEA.*
4. *To use the proposed EAPF for real-time robot navigation, with moving obstacles and for moving goal positions.*

As the EAPF is inspired by the natural potential field, it is assumed that the robot moves in a direction along the potential field angle. Figure 5.4 shows the resultant vector at any point in the field, which is the result of a repulsive force from the obstacle and an attractive force towards the goal.

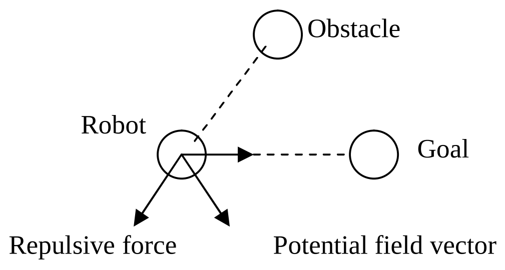


Figure 5.4 Resultant Vector direction in EAPF
(After Vadakkepat et al.)

The evolutionary algorithm, which is used to optimize the obstacle potential field function, is associated with the following steps.

1. Designing a standard attractive force function for the goal point and repulsive force functions with tuneable parameters for the different obstacles.
2. Designing potential field cost functions for the system.
3. Using the MOEA to optimise the parameters for each of the potential field functions of the obstacles, where the MOEA has four operations: selection, crossover, mutation and ranking.
4. Using the EAPF to navigate the robot.

Capi et al. proposed a Radial Basis Function Neural Network (RBFNN) for real-time gait synthesis of a humanoid robot for climbing stairs [19] and compared the results with a traditional GA implementation. The block diagram of the proposed GA method is shown in figure 5.5a. An initial population is generated, based on the initial conditions and the range of system variables. The angle trajectory is presented as a polynomial of time. Inverse dynamics is used to calculate the torque vector. The GA selects the best individual which meets the criterion after a maximum number of generations. Finally, the gait is generated for minimum CE (Consumed Energy) and TC (Torque Change), based on the result of the GA.

In real-time operation, the time used to get the minimum CE and TC has to be as small as possible. In order to achieve this, the RBFNN is taught using the GA's results. Figure 5.5b shows the structure of the RBFNN. In the network, the input neurons are connected to the environment. The hidden layer applies a Gaussian transformation function from the input to the hidden data space (which is of high dimensionality). The output layer shows a linear response to the applied input signals.

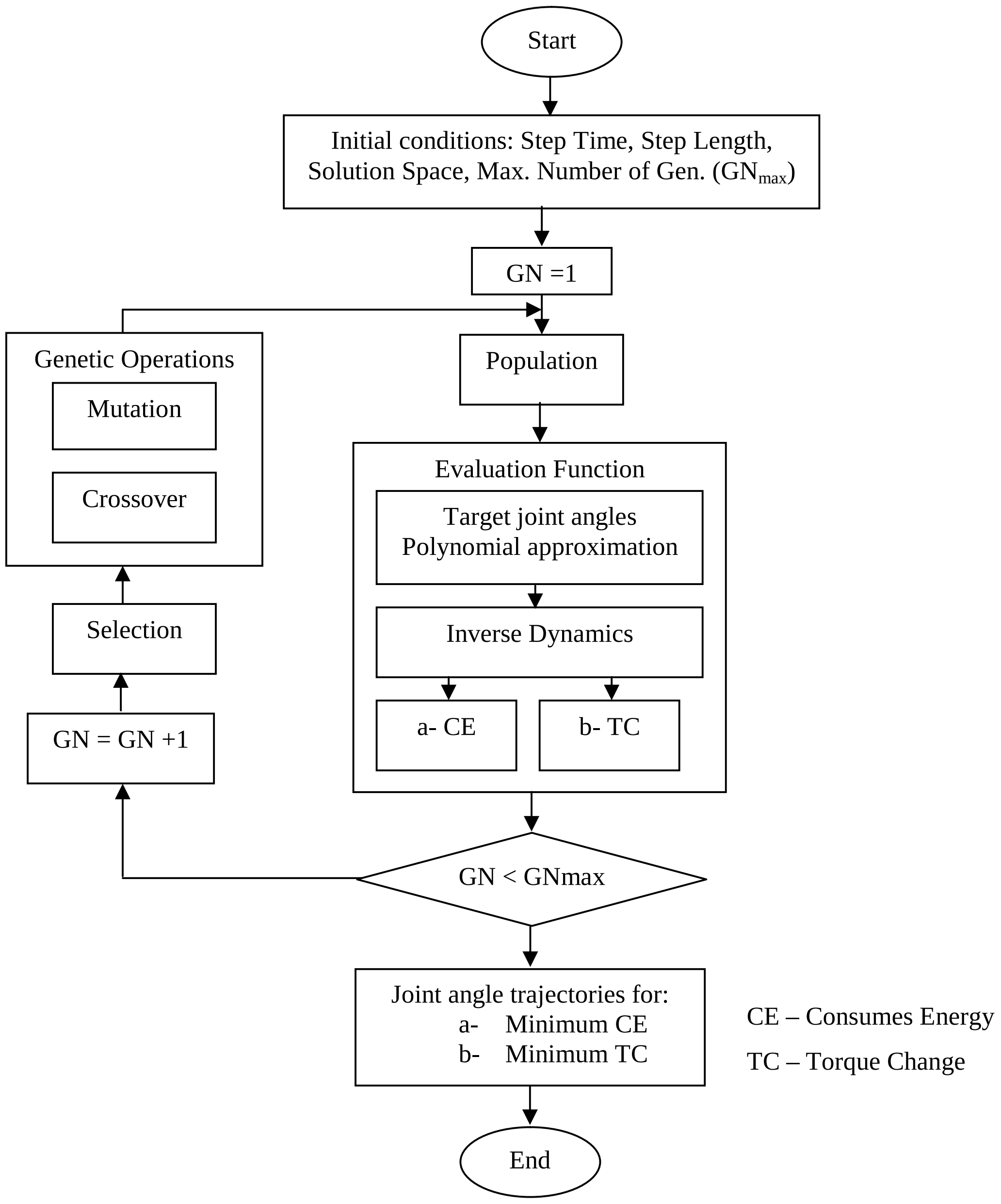

Start
Initial conditions: Step Time, Step Length, Solution Space, Max. Number of Gen. (GNmax)
GN =1
Genetic Operations
Mutation
Crossover
Population
Evaluation Function
Target joint angles Polynomial approximation
Inverse Dynamics
a- CE
b- TC
Selection
GN = GN +1
GN < GNmax
Joint angle trajectories for: a- Minimum CE b- Minimum TC
CE – Consumes Energy
TC – Torque Change
End

Figure 5.5a Block Diagram of Proposed Method Using GA

According to the simulation results, the RBFNN was able to perform as efficiently as a GA. Capi et al. concluded that, 1) the stability of the optimal gait generated is important, 2) when minimum CE is used as the cost function the robot shows a straighter posture, 3) there is a 40% energy reduction when the minimum CE is used as a cost function and 4) the RBFNN gives good results in a real-time implementation.

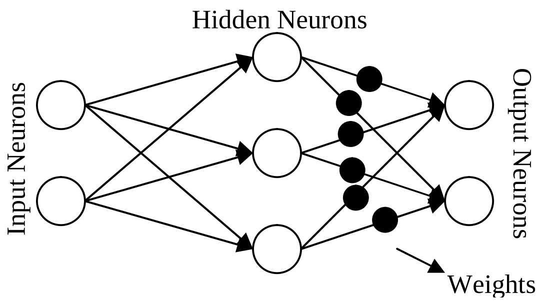
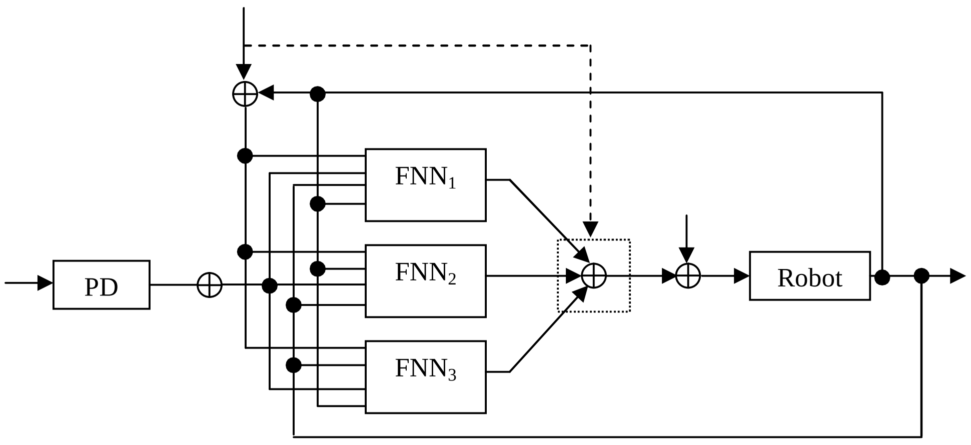


Figure 5.5b Structure of RBFNN

(After Capi et al.)

5.4 Other Methods used to Control ANNs and Robotics

Patińo et al. proposed a methodology for adaptive motion control, based on neural networks originally used for robot manipulators [20]. Figure 5.6 shows a Neural Network based adaptive controller.



FNN$_i$ is ith Feed-Forward Neural Network – Neural network bank.

Figure 5.6 Feedback robust Adaptive Control System for Robot

(After Patińo et al.)

The control structure used is an inverse-dynamics model. It is trained off-line with each NN (referred to as a FNN in figure 5.6) representing a specified payload condition. The coefficients of a linear combination of neural network outputs and uncertain payloads are adjusted using a stable controller-parameter adjustment mechanism. This helps to reduce the computational time and hence the adaptation to changes in the robot parameters is faster. The neural network, during its learning process, produces as error called the control error. This control error converges asymptotically to the neighbourhood of zero. The size of the error is evaluated and depends on the approximations to error degree given by the neural network bank. The experiments were conducted on a PUMA-560 model robot and the results showed good performance with the proposed method.

A biologically inspired neural controller for a quadruped robot was introduced by Billard and Ijspreet [21]. The neural controller technique proposed was shown to generate patterns for gait production, which allows a continuous passage from walking to trotting and then to galloping. It also showed the control of sitting and lying behaviour. The neural network was developed with oscillators composed of leaky-integrator neurons. The neurons control pairs of flexor-extensor muscles, attached to each joint. In proportion to the contraction of simulated muscles and joint flexors, the neural network receives sensory feedback. Using a model known to operate in cats, the robot locomotion is activated by either applying a tonic (non-oscillating) input or by sensory feedback from the extending legs.

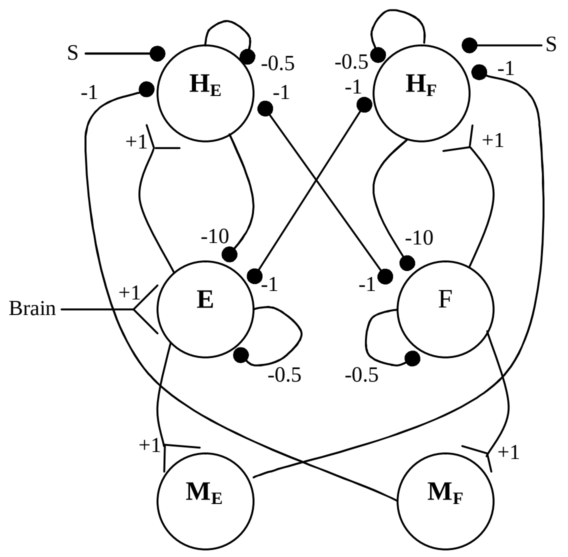


Figure 5.7 Connectivity within one oscillator

(After Billard and Ijspeert)

Brain – input tonic

M_E and M_F – motor-neurons for the extensor and flexor muscles

E, F, HE and HF – inter-neurons

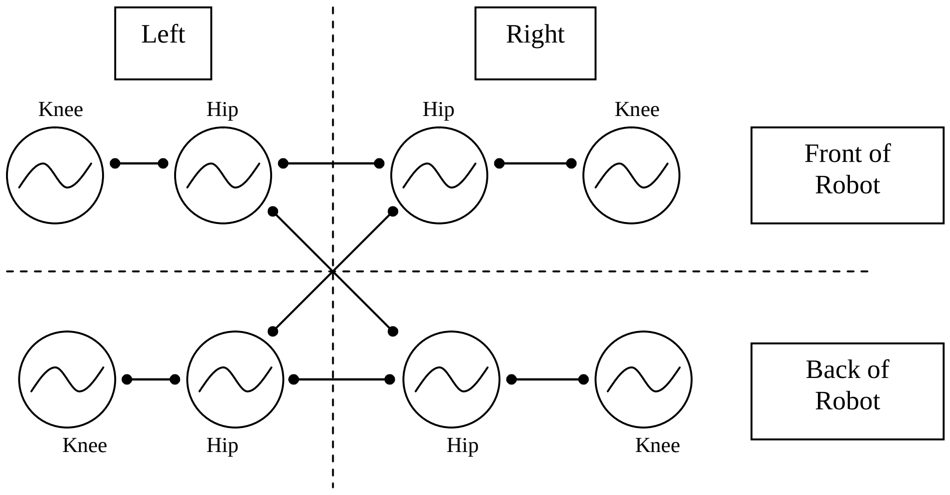


Figure 5.8 Coupling among hip and knee oscillators

(After Billard and Ijspeert)

The oscillator has four inter-neurons and two motor-neurons as shown in figure 5.7. The connectivity between the oscillators is shown in figure 5.8. The tonic inputs for walk, trot and gallop are as follows.

1. Walk – [0.0, 0.5. 0.25, 0.75]
2. Trot – [0.0, 0.5, 0.0, 0.5]
3. Gallop – [0.0, 0.1, 0.5, 0.6]

As mentioned earlier, the gaits can also be activated using sensory feedback. Extending the legs generates sensory feedback from joint angle sensors; this is adequate to start locomotion.

5.5 Context of the Current Research

The Subsumption architecture has the sole purpose of controlling a robot at the behavioural level. The work presented in this thesis controls the robot in real-time using an EA, without complex behavioural layers. Unlike the Subsumption Architecture, the RTEA keeps changing the control parameters of the robot or robot's neural network and hence, it responds to the conditions underfoot without passing the input information from layer to layer, as Subsumption Architecture does.

Hugo de Garis's GenNets are developed to perform a specific time-dependent task. The RTEA does not train the neural network to do one particular task. Instead it is capable of controlling the neural network in many different tasks, like climbing rocky-sandy mountains, etc. As mentioned earlier, de Garis used only mutation in the training algorithm, which is similar to the RTEA; however, the RTEA controls the robot in real-time through mutation, whereas the de Garis algorithm operates off-line.

The Saridis and Degney control strategy is similarly not real-time. Yang and Meng showed a control strategy for a mobile robot without any cost function or learning method, where the motion is controlled by the local selection of neural activity. The RTEA presented in this project investigates several methods to control the motion of the robot without sticking to one particular method of control.

Likewise, Vadakkept used MOEA and EAPF to control his robot. The MOEA uses operators like selection, crossover and mutation; whereas, RTEA uses only mutation to explore the control parameter values in the search space.

Capi et al. proposed a RBFNN for real-time gait synthesis and compared the results with a traditional GA based method. The RTEA controls the robot without any preliminary training or a fixed set of trained data to select from.

Other control methods by Billard, Ijspreet and Patińo et al. showed different attempts to control ANNs and Robots. The RTEA developed in this project is unique because the RTEA operators were investigated with a wide range of control situations. This helped to give insight into the capacity of the RTEA to control the robot in real-time.

It may seem that the work illustrated in the sections above represents a wide cross-section of research. However, they are all attempts to adaptively control a robot in real-time – as is this project. Their different nature underlines the uniqueness of the work presented here as their aims represent the closest work, in the literature, to this project.

Chapter 6

Initial Investigation and System Implementation

6.1 Introduction

In this chapter, the initial investigation of the robot simulators, neuron models and network topologies used in this project are explained. As the simulation system was developed as part of the project, the first section explains why it was decided to develop an in-house solution when ready-made simulators and robots can be used. The next section explains the design of the biped simulator, the development of neuron models and network topologies. Then the results acquired from these investigations are discussed along with an explanation of the modifications made to the neuron model as a result of these. Finally, the chapter ends with a discussion of the lessons learned from the biped which were fed into the development of the Quadruped models, discussed in the next chapter.

6.2 System Implementation

The algorithms developed in this project could have been tested on some ready-made systems. For example, the AIBO robot from Sony [1], is a programmable platform, into which different robot action modules can be added. Although this resource is accessible for academic research, it is expensive, and iterative testing on the robot may cause damage. Also, another advantage of developing an in-house simulation is that it gives an insight into the system, which enhances the understanding of the operation of the algorithms tested. Trouble-shooting is also easier. These latter comments also apply to ready-made software simulators and hence it was decided to develop an in-house simulation and testing system.

6.2.1 Biped Robot System

For the initial investigation, the robot test system used was a simulation of a fully stable biped walking robot. This model was developed by McMinn [2] as part of his PhD project. It was based on a physical robot (shown below in figure 6.1) used in his experiments.

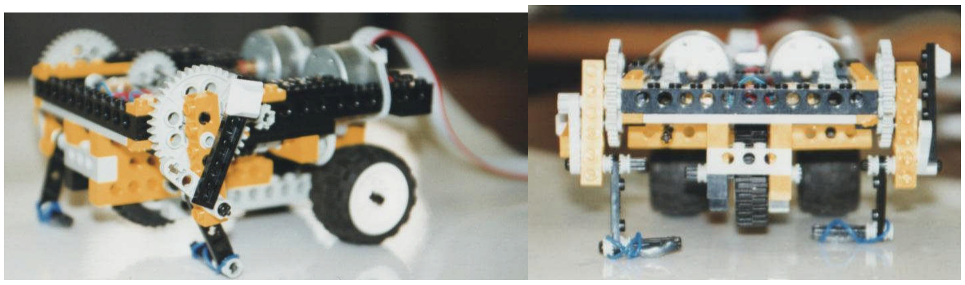


Figure 6.1 Biped Robot

Subsequently Muthuraman [3] also used this model in his work. The model was therefore well tested and has been used as the basis of several papers [4][5]. The robot’s leg may have active or passive degree(s) of freedom or both. An active degree of freedom is one which is actuator operated – generally either moving the leg clockwise or anticlockwise. The actuator used in this project is a simulated linear servomotor, obeying the simple equation, shown below (equation 6.1)

$$\frac{d}{dt}\begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}0 & 1 \\ 0 & 0\end{bmatrix}\begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix} + \begin{bmatrix}0 \\ 1\end{bmatrix}V \qquad (6.1)$$

Where ‘θ’ is the angle moved and ‘V’ is the input stimulation (Voltage). For more details refer to [6]. A Passive degree of freedom is where further leg movement may be controlled by an active degree of freedom (by means of another joint) and (or) by the physical design of the robot (through gravity). Figure 6.2 below may help to visualize the operation of the active and passive degrees of freedom.

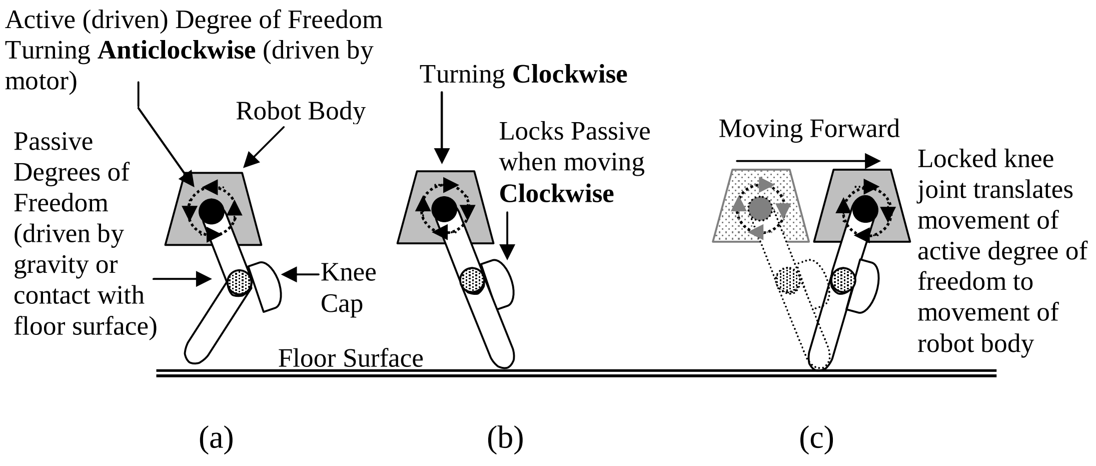

Active (driven) Degree of Freedom Turning **Anticlockwise** (driven by motor)
Robot Body
Passive Degrees of Freedom (driven by gravity or contact with floor surface)
Knee Cap
Floor Surface
Turning **Clockwise**
Locks Passive when moving **Clockwise**
Moving Forward
Locked knee joint translates movement of active degree of freedom to movement of robot body
(a)
(b)
(c)

Figure 6.2 Active and Passive Degrees of Freedom

Figure 6.2 (a) shows the active degree of freedom turning anticlockwise, making the leg move forward (from left to right in the figure). During this period, the passive degree of freedom is loose and remains so as long the active degree turns anticlockwise. Figure 6.2 (b) shows the active degree turning clockwise and in this case the passive degree gets locked by the knee. This causes the robot to progress forward as shown in figure 6.2 (c). Thus the passive degree of freedom depends on the active degree and the physical design of the robot.

The simulation of the biped robot used in this project has an active degree of freedom on the hip joints and a passive degree of freedom at each knee joint. This is shown in figure 6.3. It operates as explained above. In the robot simulators, the robot's leg movement was restricted. It was calibrated from position 0 (zero) to position 30; although this is slightly different from the physical servomotor positions, the positions calibrated resemble the electro-mechanical restriction of the real robot. Position 10 was assumed as touch down (on the surface) for forward movement and position 20 as release of knee. This is shown in figure 6.4.

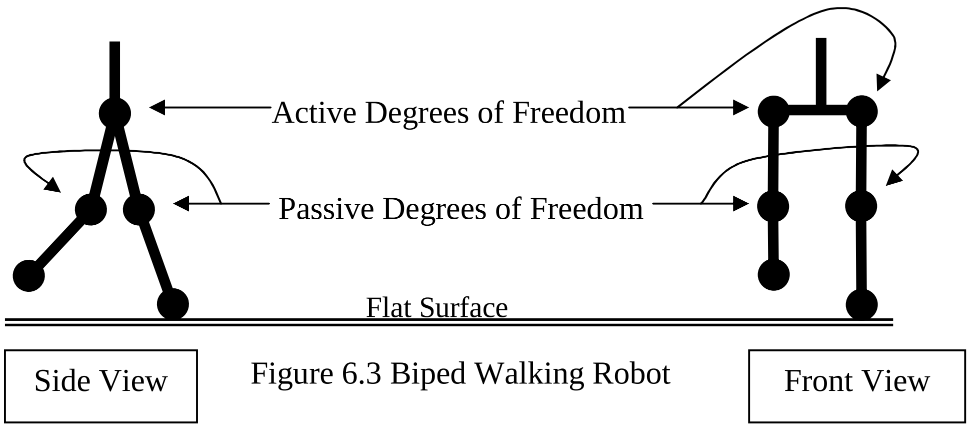


Figure 6.3 Biped Walking Robot

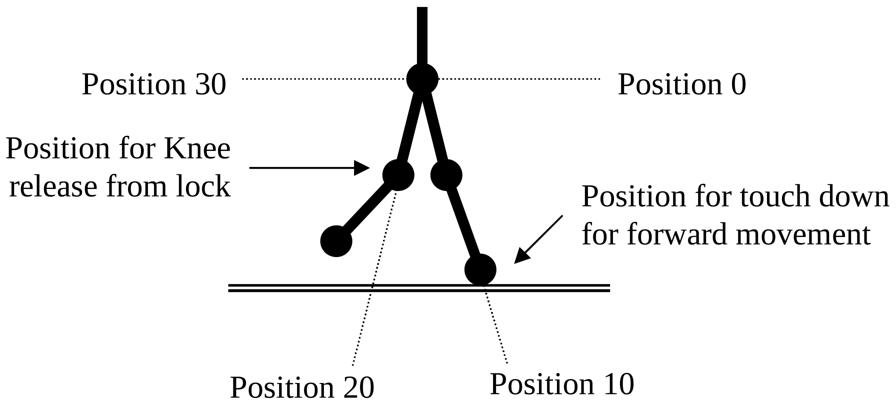

Position 30
Position 0
Position for Knee release from lock
Position for touch down for forward movement
Position 20
Position 10

Figure 6.4 Robot Leg Positions

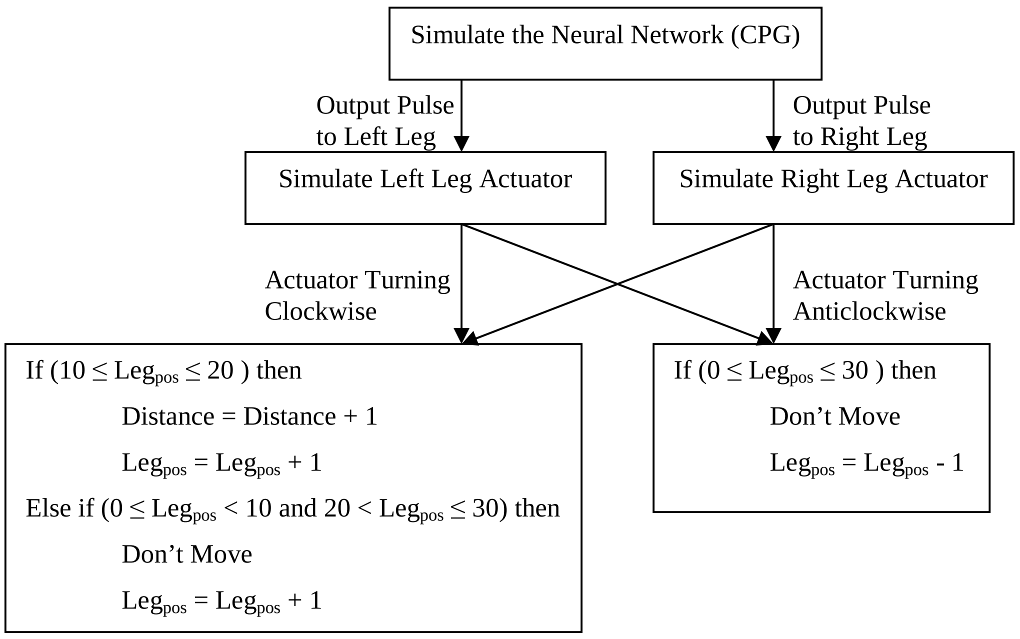
The robot's leg was designed to move one unit clockwise on receiving one unit of positive pulse from the controlling network and move one unit anticlockwise on receiving one unit of negative pulse. While the leg moves anticlockwise, the body of the robot does not move forward or backward because the knee is released (as explained above) and the leg is not in contact with the floor. The body advances forward by one unit, when the leg moves clockwise by one unit, on the condition that the leg is touching the flat surface. In the simulator, the leg is assumed to be touching the flat surface between position 10 and 20 (refer to figure 6.4). The equations below show the relationship between the robot's body movement (distance moved), leg positions and actuator movement at a given time.

$$\begin{aligned}
&If\,(Leg_{act} \xleftarrow{\text{Re}ceives} +1 \;\; and \;\; 10 \le Leg_{pos} \le 20) \\
&\Rightarrow d_t = d_t + 1 \;\; and \;\; Leg_{pos} = Leg_{pos} + 1 \\
&If\,(Leg_{act} \xleftarrow{\text{Re}ceives} +1 \;\; and \;\; 0 \le Leg_{pos} < 10 \;\; or \;\; 20 < Leg_{pos} \le 30) \\
&\Rightarrow d_t = d_t \;\; and \;\; Leg_{pos} = Leg_{pos} + 1 \\
&If\,(Leg_{act} \xleftarrow{\text{Re}ceives} -1 \;\; and \;\; 0 \le Leg_{pos} \le 30) \\
&\Rightarrow d_t = d_t \;\; and \;\; Leg_{pos} = Leg_{pos} - 1
\end{aligned} \qquad (6.2)$$

where d_t is the Distance at 't' time
Leg_{pos} is the Leg Position
Leg_{act} is the Leg Actuator

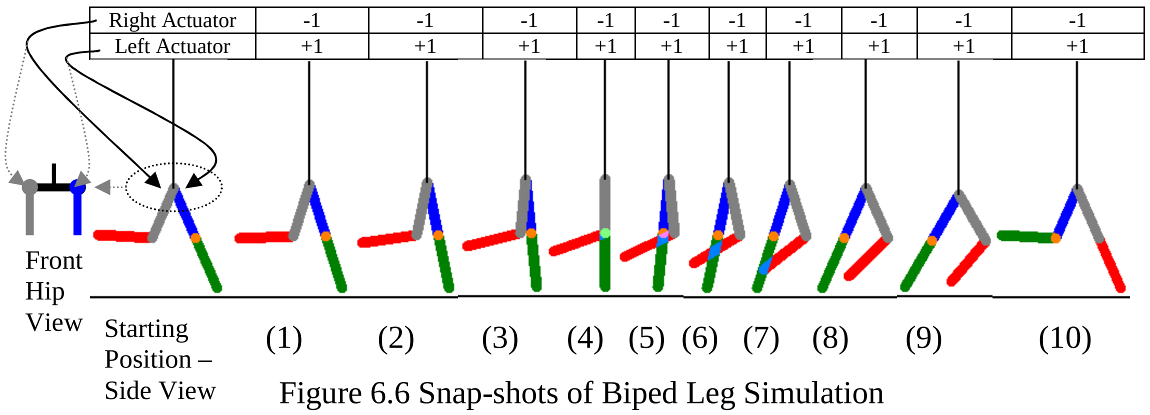
In order to get an efficient normal bipedal walking motion, in this simulation, one of the robot's actuators has to receive a stream of ten units of positive pulse while the other actuator has to receive a stream of ten units of negative pulse. More about the pulse generator (the Central Pattern Generator) is discussed in the next section. The robot's simulator was coded (as was all the other programming in the project) using the Borland C++ Builder. The method of simulation is given in figure 6.5.

The snap-shots (shown in figure 6.6), of the biped robot leg simulation may help in visualizing the robot's body movement as the actuator of one leg receives positive pulses and the other one receives negative pulses. Note that in the simulation, the robot is stable (shown above as a black string suspended from the ceiling connected to the robot’s hip joints). Hence, in this project, only generating the correct gait pattern was considered (the original physical robot on which this simulation was based was also fully stable).



Simulate the Neural Network (CPG)
Output Pulse to Left Leg
Output Pulse to Right Leg
Simulate Left Leg Actuator
Simulate Right Leg Actuator
Actuator Turning Clockwise
Actuator Turning Anticlockwise
If ($10 \leq Leg_{pos} \leq 20$) then
Distance = Distance + 1
$Leg_{pos} = Leg_{pos} + 1$
Else if ($0 \leq Leg_{pos} < 10$ and $20 < Leg_{pos} \leq 30$) then
Don’t Move
$Leg_{pos} = Leg_{pos} + 1$
If ($0 \leq Leg_{pos} \leq 30$) then
Don’t Move
$Leg_{pos} = Leg_{pos} - 1$

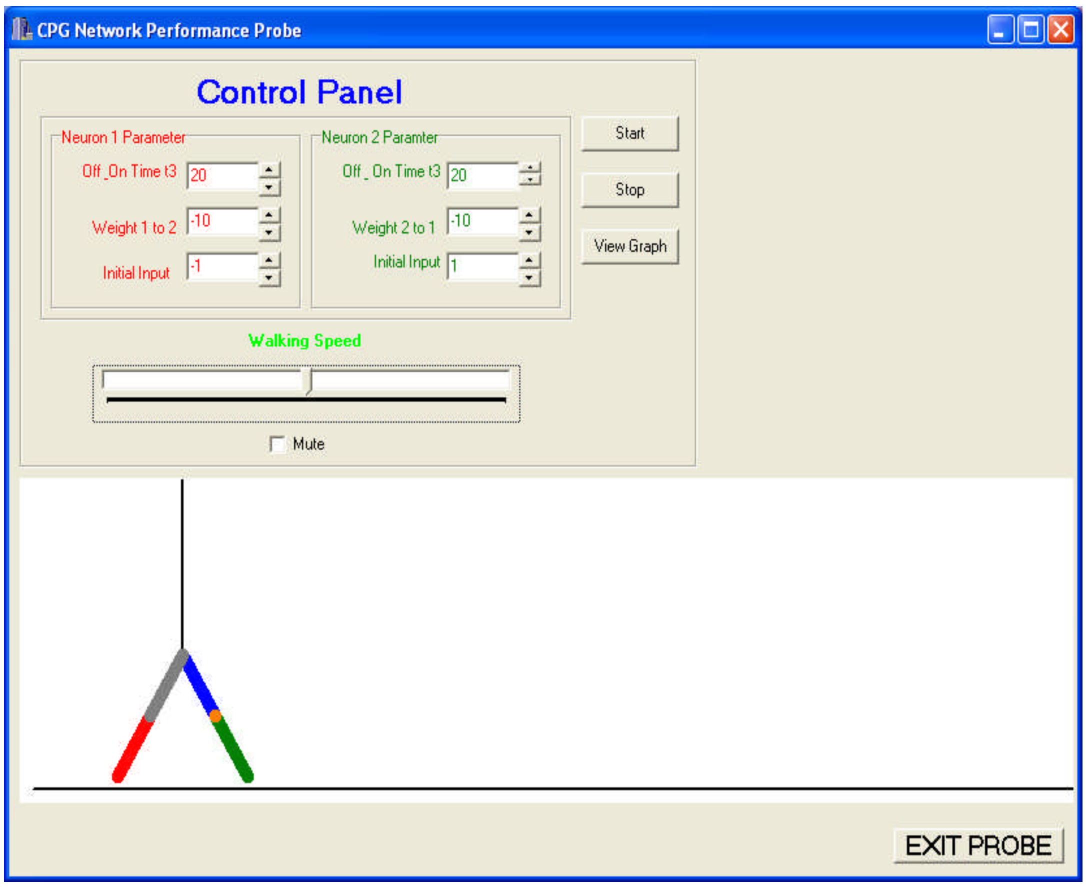
Figure 6.5 C++ Simulation Method



Right Actuator	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Left Actuator	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1

Figure 6.6 Snap-shots of Biped Leg Simulation

In the above figure, from the starting position, the generated pulses are fed to the actuators. The bent leg (knee released) receives positive pulses from time-frame 1 to 10 and the other leg receives negative pulses. In each time-frame, one unit of each pulse is fed to the actuators and, as the pulses are sequential, the gait for walk can be observed from time frame 1 to 10. Figure 6.7 is a snap-shot of the program developed in-house to investigate the biped network operating in real-time.



Neuron 1 Parameter
Off _On Time t3 20
Weight 1 to 2 -10
Initial Input -1

Neuron 2 Paramter
Off _ On Time t3 20
Weight 2 to 1 -10
Initial Input 1

Figure 6.7 Snap-Shot of Biped Simulator

6.2.2 Investigation of CPG Networks for Bipeds

As explained in chapter 2, McMinn showed that time-dependent neuron models perform better than McCulloch-Pitts models in applications requiring timing control. Hence, it was decided to use the proven model for this investigation. However, unlike McMinn's leaky integrator neuron model, a new time-dependent neuron model with similar time-dependent behaviour was developed. McMinn's neuron model output is set to "+1" (positive pulse) for a certain period of time if the neuron's membrane potential is above the internal threshold and at all other times the output is set to zero. The new model fires a series of positive pulses for certain period of time, called the "On-time" and a series of negative pulses for a certain period of time, called the "Off-time" (both fixed by the EA). With the help of the positive pulses, the robot's actuator can move clockwise and with the help of the negative pulses it can move anticlockwise (refer to figure 6.6). Using the new model's properties, the number of pulses fired is controlled and so are the actuators of the robot legs. If McMinn's neuron model were to be used, it would only have moved the actuator clockwise if the output of the neuron were "+1" and in all the other time-steps it would not do anything. Of course, with some modification to the current actuator, McMinn's model could have been used; however, as the new model fitted the current robot simulation better than McMinn's model, the investigation proceeded with the new model. The initial neuron model had the following properties (all are evolvable parameters). The possible output of such a neuron is shown in figure 6.8.

Neuron Parameters

T_{ON} – On Time

T_{OFF} – Off Time

θ – Threshold

W – Weight (The number of weights depend on the number of neurons connected)

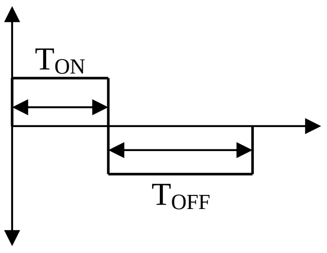


Figure 6.8 Sample Neuron Output

When the neuron receives a Net (Sum of products of inputs and weights) value greater than the threshold (θ), it fires a unit of positive pulse for the T_{ON} time. Once T_{ON} is up, it then fires a unit of negative pulse for T_{OFF}. If the Net value is less than the threshold, it fires a negative pulse continuously, until the Net value gains enough strength to overcome the threshold. The neuron activation function is shown in equation 6.3.

$$\begin{aligned}
&if\,(Net_j(t) > \theta_j) \Rightarrow for \quad 0 < t < T_{ONj} \quad Out_j(t) = +1 \\
&and \qquad\qquad\quad \Rightarrow for \quad 0 < t < T_{OFFj} \quad Out_j(t) = -1 \\
&if\,(Net_j(t) < \theta_j) \Rightarrow for \quad 0 < t < \infty \quad\; Out_j(t) = -1 \\
&\quad where \quad Net_j = \sum_{i=1}^{n} In_i \times W_i \\
&\qquad\qquad T_{ONj} - Number \;\; of \;\; ON \;\; Time-Steps \;\; set \;\; by \;\; EA \\
&\qquad\qquad T_{OFFj} - Number \;\; of \;\; OFF \;\; Time-Steps \;\; set \;\; by \;\; EA
\end{aligned} \qquad (6.3)$$

Where Net_j is the sum of product of input and weight – Neuron potential of j^{th} neuron

θ_j is the threshold of the j^{th} neuron

Out_j is the output of j^{th} neuron

In_i is the i^{th} input to the j^{th} neuron

W_i is the i^{th} weight of the j^{th} neuron

T_{ONj} is the On time of the j^{th} neuron

T_{OFFj} is the Off time of the j^{th} neuron

During this ON/OFF cycle, the neuron does not consider any changes made to its inputs (as does a biological neuron). As the neuron is said to generate streams of positive and negative pulses, theoretically, one would expect an output which may roughly look like as shown in figure 6.9. Practically, in simulation, there is no gap between the pulses as shown figure 6.10. Note: For this example, the T_{ON} and T_{OFF} is taken as 2 units. It is also assumed that the neuron's Net was above the threshold and fired "+1s" for T_{ON} time and then "-1s" for T_{OFF} time.

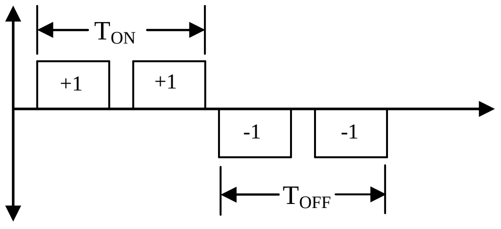


Figure 6.9 Theoretical Neuron Output form

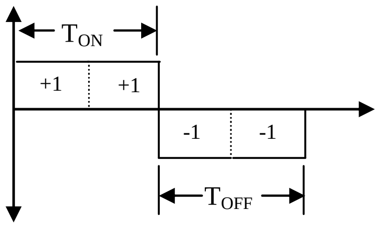


Figure 6.10 Simulated Neuron Output form

Using this neuron model, an investigation was started using a recurrent network topology. This is explained in the next section.

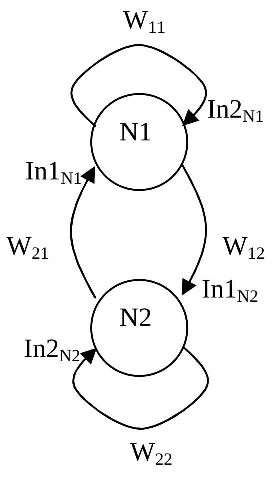
6.2.2.1 Recurrent Network

To operate a biped as shown in section 6.2.1 with the neuron type described in section 6.2.2, at least two neurons are required. A recurrent network using the above neuron model was heuristically selected for an initial investigation. The network is fully connected and has two neurons as shown in figure 6.11.

The simulation method of the network as described is given below.

1. The initial input from N2 to N1 ($In1_{N1}$) and N1 to N1 ($In2_{N1}$) is set to "-1"
2. The initial Input from N1 to N2 ($In1_{N2}$) and N2 to N2 ($In2_{N2}$) is set to "+1". The Net values of N1 and N2 are calculated (Sum of products of inputs and weights).
3. If the Net is above the threshold (θ), the neuron fires a positive pulse for T_{ON} time and then a negative pulse for T_{OFF} time.
4. If the Net is below the threshold, the neuron fires a negative pulse until Net moves above the threshold.

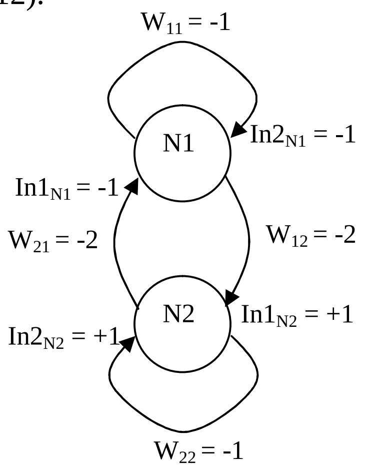
5. Steps 3, 4 and 5 are repeated for the number of Time-Steps (the term used in simulation, referring to the number of times of repeating the function) the user specifies.



Where N1 is neuron 1
N2 is neuron 2
W_{11} is recurrent connection weight of N1
W_{22} is recurrent connection weight of N2
W_{12} is weight between N1 and N2
W_{21} is weight between N2 and N1
$In1_{N1}$ is the initial input from N2 to N1
$In2_{N1}$ is the initial input from N1 to N1
$In1_{N2}$ is the initial input from N1 to N2
$In2_{N2}$ is the initial input from N2 to N2

Figure 6.11 Recurrent CPG Network

The operation of the network is illustrated with an example below. (Refer to figure 6.12).



N1:
$T_{ON} = 2$
$T_{OFF} = 2$
$\theta = 0.5$

N2:
$T_{ON} = 2$
$T_{OFF} = 2$
$\theta = 0.5$

Figure 6.12 Recurrent Network – Example

The step by step operation of the network is given below, with the simulated output of the network shown in figure 6.13.

Note that the Net values of both the neurons were evaluated at the same time.

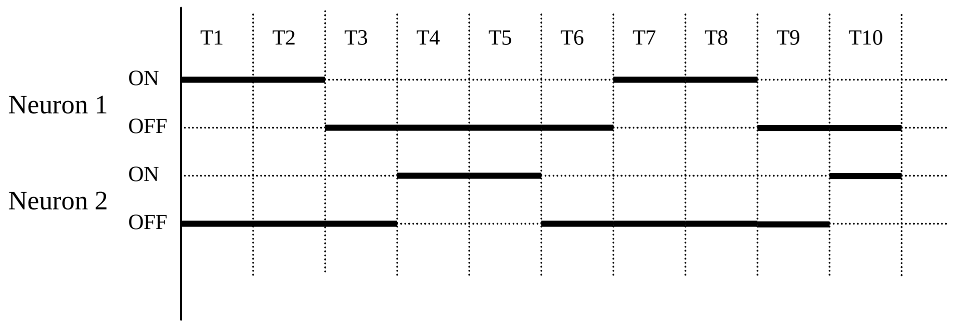


Figure 6.13 Recurrent Network Simulation Output

T1	<u>Neuron 1</u> $Net = (In1_{N1} \text{ x } W_{21}) + (In2_{N1} \text{ x } W_{11})$ $= (-1 \text{ x } -2) + (-1 \text{ x } -1)$ $= +3$ $Net (+3) > \theta (0.5) \Rightarrow Out = +1$ for $0 < t < T_{ON}$ ($T_{ON} = 2$ – for two Time Steps)	<u>Neuron 2</u> $Net = (In1_{N2} \text{ x } W_{12}) + (In2_{N2} \text{ x } W_{22})$ $= (1 \text{ x } -2) + (1 \text{ x } -1)$ $= -3$ $Net (-3) < \theta (0.5) \Rightarrow Out = -1$ until $Net > \theta$
	Out of N1 is input $In1_{N2}$ (of neuron 2) and $In2_{N1}$ (of neuron 1) $\rightarrow In1_{N2} = +1$ and $In2_{N1} = +1$ Out of N2 is input $In1_{N1}$ (of neuron 1) and $In2_{N2}$ (of neuron 2) $\rightarrow In1_{N1} = -1$ and $In2_{N2} = -1$	
T2	<u>Neuron 1</u> No change in N1 as it is performing ON cycle	<u>Neuron 2</u> (New) $Net = (In1_{N2} \text{ x } W_{12}) + (In2_{N2} \text{ x } W_{22})$ $= (+1 \text{ x } -2) + (+1 \text{ x } -1)$ $= -3$ $Net (-3) < \theta (0.5) \Rightarrow Out = -1$ until $Net > \theta$
	Out of N1 is input $In1_{N2}$ (of neuron 2) and $In2_{N1}$ (of neuron 1) $\rightarrow In1_{N2} = +1$ and $In2_{N1} = +1$ Out of N2 is input $In1_{N1}$ (of neuron 1) and $In2_{N2}$ (of neuron 2) $\rightarrow In1_{N1} = -1$ and $In2_{N2} = -1$	
T3	<u>Neuron 1</u> ON cycle finished, now in OFF cycle $\Rightarrow Out = -1$ for $0 < t < T_{OFF}$ ($T_{OFF} = 2$ – for two Time Steps)	<u>Neuron 2</u> $Net = (In1_{N2} \text{ x } W_{12}) + (In2_{N2} \text{ x } W_{22})$ $= (+1 \text{ x } -2) + (+1 \text{ x } -1)$ $= -3$ $Net (-3) < \theta (0.5) \Rightarrow Out = -1$ until $Net > \theta$
	Out of N1 is input $In1_{N2}$ (of neuron 2) and $In2_{N1}$ (of neuron 1) $\rightarrow In1_{N2} = -1$ and $In2_{N1} = -1$ Out of N2 is input $In1_{N1}$ (of neuron 1) and $In2_{N2}$ (of neuron 2) $\rightarrow In1_{N1} = -1$ and $In2_{N2} = -1$	
T4	<u>Neuron 1</u> No change in N1 as it is performing OFF cycle	<u>Neuron 2</u> (New) $Net = (In1_{N2} \text{ x } W_{12}) + (In2_{N2} \text{ x } W_{22})$ $= (-1 \text{ x } -2) + (-1 \text{ x } -1)$ $= +3$ $Net (+3) > \theta (0.5) \Rightarrow Out = +1$ for $0 < t < T_{ON}$ ($T_{ON} = 2$ – for two Time Steps)
	Out of N1 is input $In1_{N2}$ (of neuron 2) and $In2_{N1}$ (of neuron 1) $\rightarrow In1_{N2} = +1$ and $In2_{N1} = +1$ Out of N2 is input $In1_{N1}$ (of neuron 1) and $In2_{N2}$ (of neuron 2) $\rightarrow In1_{N1} = -1$ and $In2_{N2} = -1$	

Similar calculations were carried out for the rest of the Time-Steps and the output of the network is shown in figure 6.13. The network was coded and simulated for 10 Time-Steps. The output is shown in figure 6.14

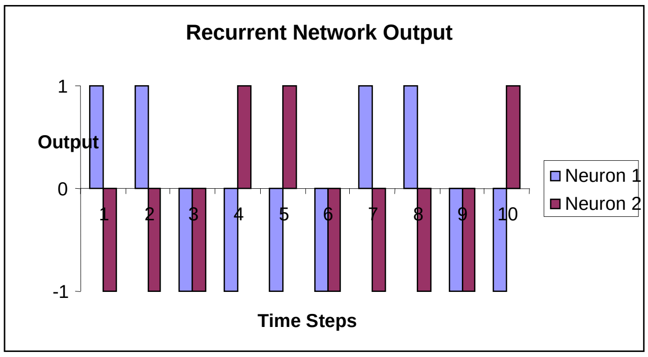


Figure 6.14 Recurrent Network Simulated Output.

Normally with T_{ON} = 2 and T_{OFF} = 2 for both the neurons, one would intuitively expect a graph similar to that shown in figure 6.15

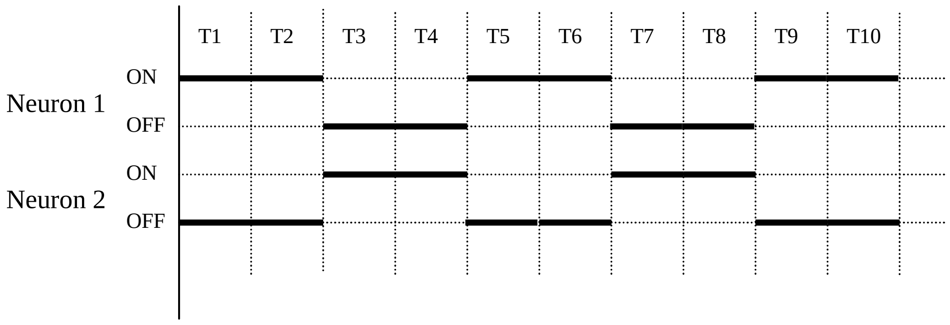


Figure 6.15 Expected Output

The graph in figure 6.15 shows the expected symmetrical output based on the parameter values of T_{ON} and T_{OFF} of neuron-1 and neuron-2. The lack of symmetry observed in figure 6.13 is due to the state-change of the neuron and its corresponding output. For example, in the transition between states ON to OFF of neurons between T2 to T3, the neuron-1 finishes its ON cycle on T2 and changes to the OFF state in T3. The output of neuron-1 in T2 is +1 and this output is the input of the neuron-2 in T3. Hence, in T3, neuron-2’s Net moves above the threshold and starts its cycle, whereas one might expect neuron-2 to start its cycle at T2, as shown in figure 6.15. This lack of symmetry was investigated and the neuron model was modified. This issue is discussed in detail in the next section.

6.2.2.2 Neuron Modification

As discussed in the last section, the change of state of the neuron was carefully investigated. It was found that, during the transition of the state, the output of the neuron, which acts as the input to the following neuron in the next Time-Step, is multiplied by '-1' while the output remains as it should be. The transition of state of the same network shown in figure 6.12 is as shown below. The first three Time-Step simulation is shown below in figure 6.16.

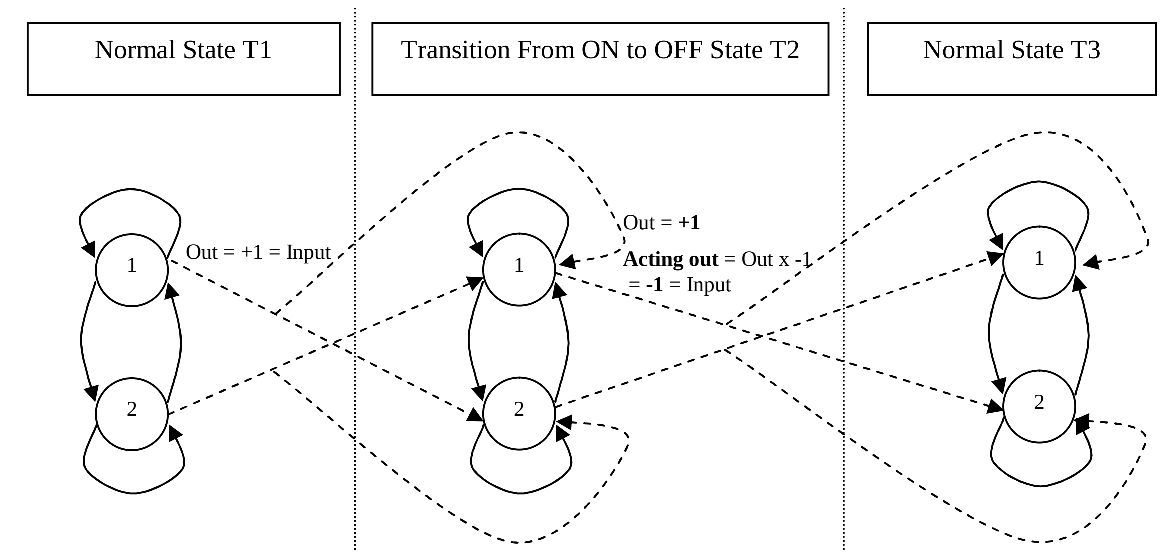


Figure 6.16 New Acting Output Model Simulation

T1	<u>Neuron 1</u> Net = ($In1_{N1}$ x W_{21}) + ($In2_{N1}$ x W_{11}) = (-1 x -2) + (-1 x -1) = +3 Net (+3) > θ (0.5) $\Rightarrow$ Out = +1 for $0 < t < T_{ON}$ (T_{ON} = 2 – for two Time Steps)	<u>Neuron 2</u> Net = ($In1_{N2}$ x W_{12}) + ($In2_{N2}$ x W_{22}) = (1 x -2) + (1 x -1) = -3 Net (-3) < θ (0.5) $\Rightarrow$ Out = -1 until Net > θ
	Out of N1 is input $In1_{N2}$ (of neuron 2) and $In2_{N1}$ (of neuron 1) → $In1_{N2}$ = +1 and $In2_{N1}$ = +1 Out of N2 is input $In1_{N1}$ (of neuron 1) and $In2_{N2}$ (of neuron 2) → $In1_{N1}$ = -1 and $In2_{N2}$ = -1	
T2	<u>Neuron 1</u> No change in N1 as it is performing ON cycle No change in N1 as it is finishing ON cycle $\Rightarrow$ Out = +1 As it is going to change it state from ON to OFF $\Rightarrow$ Acting Out = Out * -1 = +1 * -1 = **-1**	<u>Neuron 2</u> (New) Net = ($In1_{N2}$ x W_{12}) + ($In2_{N2}$ x W_{22}) = (1 x -2) + (1 x -1) = -3 Net (-3) < θ (0.5) $\Rightarrow$ Out = -1 until Net > θ
	Acting Out of N1 is input $In1_{N2}$ (of neuron 2) and $In2_{N1}$ (of neuron 1) → $In1_{N2}$ = -1 and $In2_{N1}$ = -1 Out of N2 is input $In1_{N1}$ (of neuron 1) and $In2_{N2}$ (of neuron 2) → $In1_{N1}$ = -1 and $In2_{N2}$ = -1	

T3	<u>Neuron 1</u> ON cycle finished, now in OFF cycle $\Rightarrow$ Out = -1 for $0 < t < T_{OFF}$ (T_{OFF} = 2 – for two Time Steps)	Net = ($In1_{N2}$ x W_{12}) + ($In2_{N2}$ x W_{22}) = (**-1** x -2) + (**-1** x -1) = +3 Net (+3) > θ (0.5) $\Rightarrow$ Out = -1 for $0 < t < T_{ON}$ (T_{ON} = 2 – for two Time Steps)
	Out of N1 is input $In1_{N2}$ (of neuron 2) and $In2_{N1}$ (of neuron 1) $\rightarrow$ $In1_{N2}$ = -1 and $In2_{N1}$ = -1 Out of N2 is input $In1_{N1}$ (of neuron 1) and $In2_{N2}$ (of neuron 2) $\rightarrow$ $In1_{N1}$ = -1 and $In2_{N2}$ = -1	

As shown in figure 6.16 in the transition state T2, the output is multiplied by '-1' and hence a new property was introduced to the neuron model, called the "Acting-Output". This acts as a memory parameter in the simulation. Thus by introducing this new property, "Acting-Output", the lack of symmetry in the network output was avoided. The network was simulated and the output was generated for 10 Time-Steps. The output graph is shown in figure 6.17

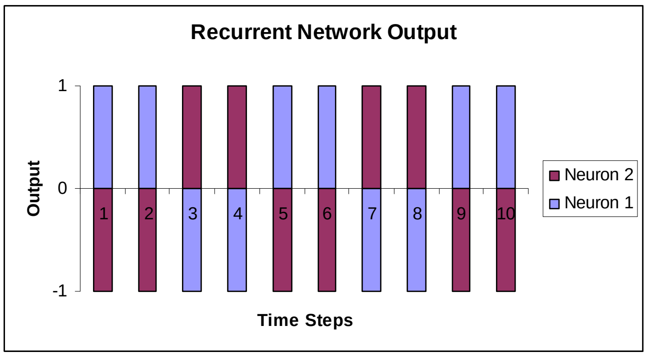
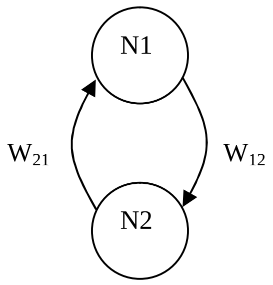


Figure 6.17 Recurrent Network Simulated Output

After improving the predictability of the neuron model, another type of network topology was investigated to check if it had any advantages over the recurrent network. This was the feed-forward network, discussed in the next section.

76.2.2.3 Feed-Forward Network

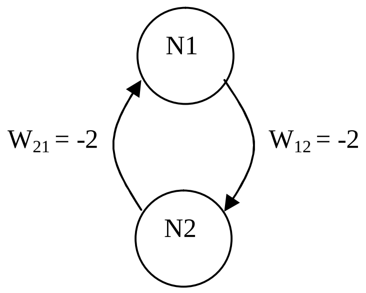
A feed-forward network for a biped has two neurons connected to each other as shown in figure 6.18. Note: This network is a recurrent network; except it has no recurrent connections. However to differentiate it from the normal recurrent network, it was decided to refer to this network as a feed-forward network.



Where N1 is neuron 1
N2 is neuron 2
W_{12} is weight between N1 and N2
W_{21} is weight between N2 and N1

Figure 6.18 Feed-Forward Network

Initially the network used the neuron model which was not modified as discussed in the last section (a neuron with NO Acting-Output). The operation of the network is given using an example below. The network parameters in the example are the same as the ones discussed in section 6.2.2.1 (refer to figure 6.12). The network and its parameters are shown in figure 6.19.



N1:	N2:
$T_{ON} = 2$	$T_{ON} = 2$
$T_{OFF} = 2$	$T_{OFF} = 2$
$\theta = 0.5$	$\theta = 0.5$

Figure 6.19 Feed Forward Network – Example

The network performed in a similar way to the recurrent network as discussed in section 6.2.2.1. Again it was simulated and the output is shown in figure 6.20.

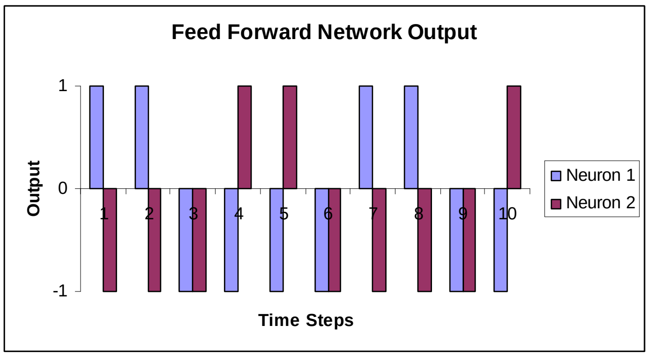


Figure 6.20 Feed-Forward Network, Simulated Output

As it showed no particular difference from the recurrent network, using the same neuron model, the modified neuron model was tried. With the same parameters shown in figure 6.19, the network was simulated for 10 Time Steps and the output is shown in figure 6.21.

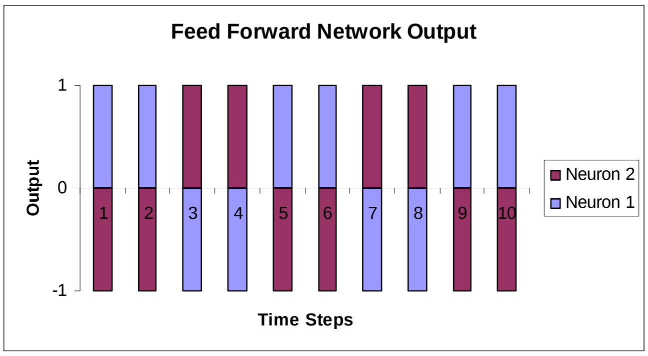


Figure 6.21 Feed-Forward Network, Simulated Output

It was found that both the networks can generate the expected patterns for a biped. The network evolve-ability was then investigated. This investigation was carried out to learn which network works the best for the biped, operated using an EA in real-time.

6.2.2.4 Network Evolvability Comparison

The network evolvability was compared by training the networks to generate patterns for the walking biped (as discussed in section 6.2.1) using an Evolutionary Algorithm. The main aim of this investigation is to learn the behaviour of the networks when their parameters are varied (that is, when the networks are operated by the EA). At this stage of the project, the RTEA was not developed. Hence, a standard GA was used to train the networks. The simulation method is shown in figure 6.22.

Each of the steps in the flow-chart shown is explained below. The chromosome population was fixed at 40. Each member of the population is a set of network parameters.

For the recurrent network, the number of parameters was ten and the chromosome is shown in figure 6.23. For the feed-forward network the number of parameters is eight and the chromosome is shown in figure 6.24. (For details of recurrent network parameters, refer to figure 6.11 and for details of feed-forward network parameters refer to figure 6.18).

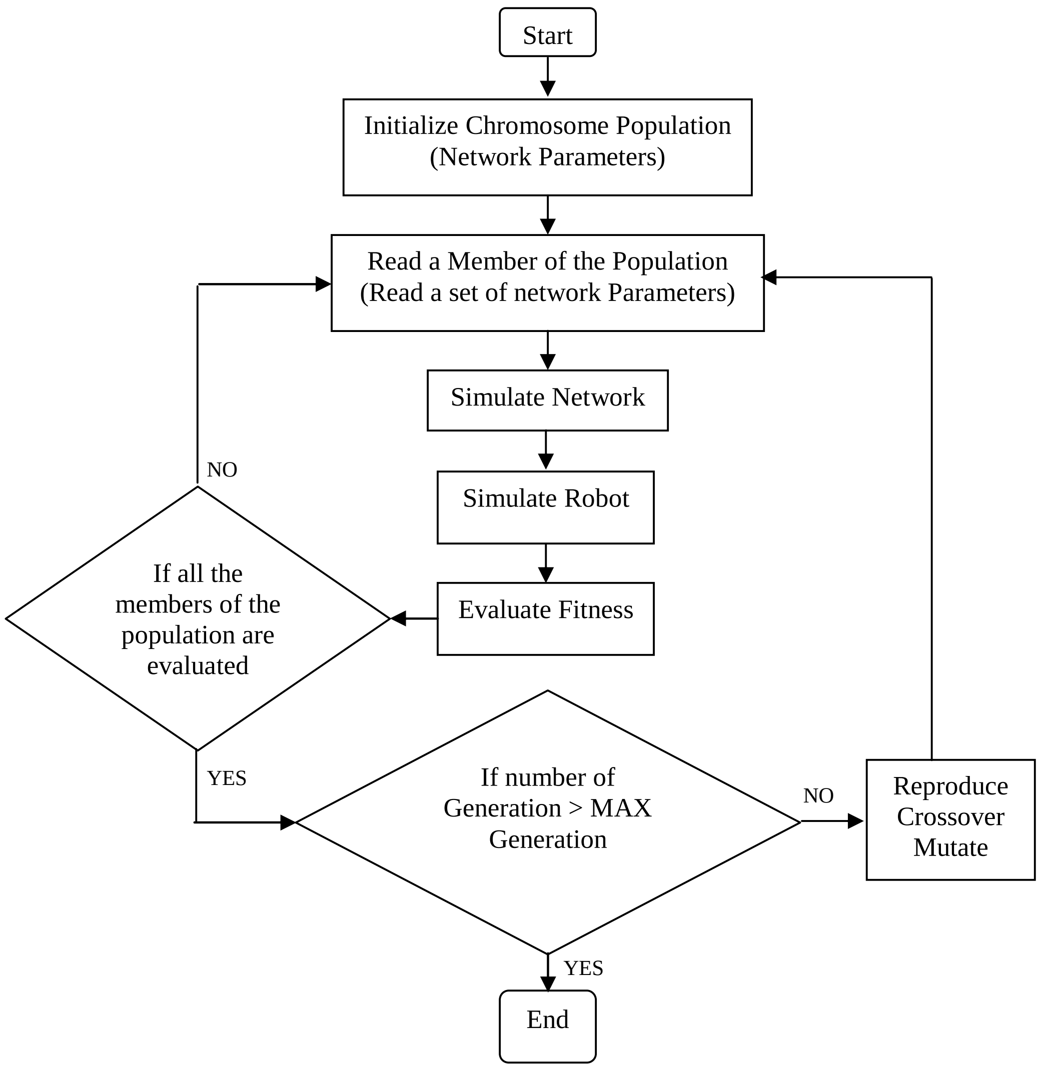

Start
Initialize Chromosome Population (Network Parameters)
Read a Member of the Population (Read a set of network Parameters)
Simulate Network
Simulate Robot
Evaluate Fitness
If all the members of the population are evaluated
NO
YES
If number of Generation > MAX Generation
NO
Reproduce Crossover Mutate
YES
End

Figure 6.22 Training Simulation Method

Neuron 1 Parameters					Neuron 2 Parameters				
T_{ON}	T_{OFF}	W_{21}	W_{11}	θ_1	T_{ON}	T_{OFF}	W_{12}	W_{22}	θ_2

Figure 6.23 Chromosome for Recurrent Network

Neuron 1 Parameters				Neuron 2 Parameters			
T_{ON}	T_{OFF}	W_{21}	θ_1	T_{ON}	T_{OFF}	W_{12}	θ_2

Figure 6.24 Chromosome for Feed-Forward Network

In the population, the ON and OFF times were randomly set between 1 and 10. The reason for this limitation is because the robot’s leg positions for walking are between

20 and 30, which only requires 10 units of positive and negative pulses. The weights and the thresholds were randomly set between 0 and 1. The random numbers were generated using Uniformly Distributed Random Numbers, as described in chapter 3, section 3.3.2.7.

As each member in the population is a set of network parameters as shown above, one member at a time is read from the population, starting from the top. Each of the members is evaluated using the fitness function. To evaluate the fitness, the network was simulated for 40 Time-Steps. The network is simulated as shown in the section 6.2.2.1, for the recurrent network and in section 6.2.2.2, for the feed-forward network. In each Time-Step the network's simulated output is fed to the robot's leg actuators. The legs are simulated as shown in section 6.2.1 and the robot's movement (distance moved – in fitness function), is governed by the equation 6.2. Apart from the distance moved, the leg error is also included in the fitness function. This means that if the leg is moving between position 0 and 10 or 20 and 30, leg error is incremented. After simulating the network and the robot for the specified number of times, the overall fitness of each chromosome is calculated using the formula shown below.

$$f(x) = \left(\left(\frac{Distance\ Moved}{TIME\_STEPS} \right) \times 100 \right) - \left(\left(\frac{Leg\quad Error}{TIME\_STEPS} \right) \times 100 \right) \qquad (6.4)$$

After calculating the fitness, the GA is applied (refer to chapter 3, section 3.3 for the operation of a GA). First, the population is reproduced, by sorting the chromosomes from most-fit to least-fit. Then the bottom half of the population is reproduced by randomly replacing each member in the bottom half of the population by a fit member from the top half.

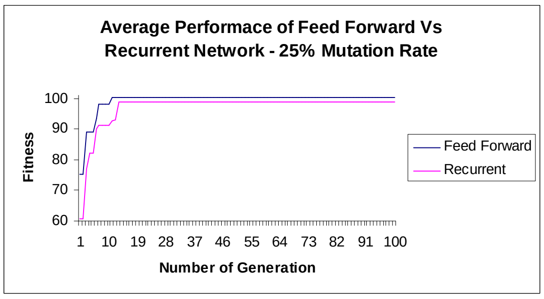
Secondly, the crossover operation was performed, by selecting each adjacent pair in the bottom half of the population, in-order to keep the fit parameters without disturbance. In this way, the new members in the bottom half will get a chance to compete with the top half in the next generation. After selecting every adjacent pair in the bottom half, a crossover point for each pair is randomly selected and the parameters are swapped.

Thirdly, the mutation rate is selected from 25%, 50%, 75% and 100%. The mutation rate represents the percentage of genes (network parameters) mutated in a chromosome. For example, for the feed-forward network the number of genes in a chromosome is eight. If the mutation rate is 25% then it shows that 2 random parameters are mutated. In every generation, the bottom half of the chromosomes are mutated according to the mutation rate selected for the experiment.

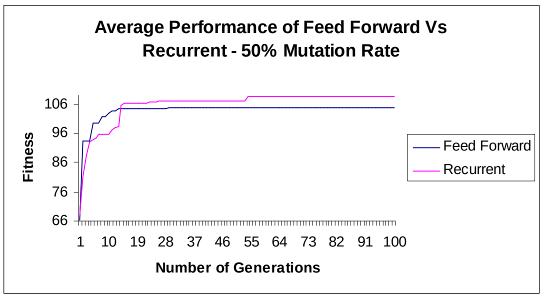
The above steps are repeated for the number of generations specified (it is 100 for the experiments conducted here). The result of the comparison of feed-forward and recurrent networks is shown below. The experimental setup information is also shown and each experiment was repeated three times with different mutation rates as shown along with the setup information.

Experiment Setup Information – Number of Trials: 3	
Feed-Forward Network 1. Evolvable parameters: Neuron 1 –N1: T_{ON1}, T_{OFF1}, W_{21} and θ_{N1} Neuron 2 – N2: T_{ON2}, T_{OFF2}, W_{12} and θ_{N2} 2. Population Size – 40 3. Number of Generations – 100 4. Time Steps – 40	**Recurrent Network** 1. Evolvable Parameters: Neuron 1 – N1: T_{ON1}, T_{OFF1}, W_{11}, W_{21}, and θ_{N1} Neuron 2 – N2: T_{ON2}, T_{OFF2}, W_{12}, W_{22} and θ_{N2} 2. Population Size – 40 3. Number of Generations – 100 4. Time Steps – 40
Mutation Rates – 25%, 50%, 75% and 100%	

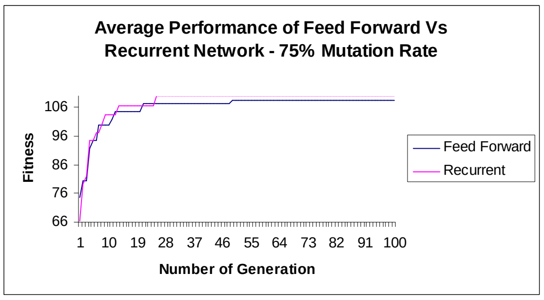
Note: the results shown below are the average output of three trials of the networks.

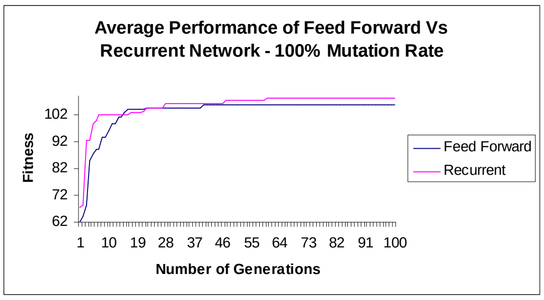


(a)



(b)

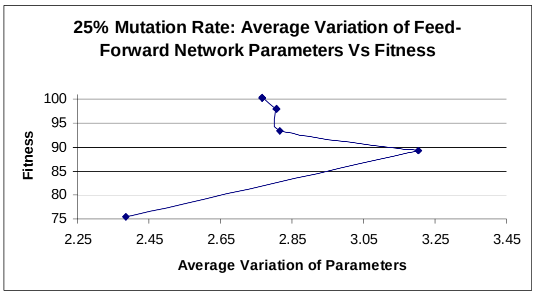


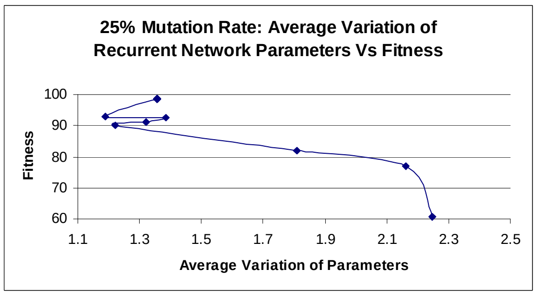


(c) (d)

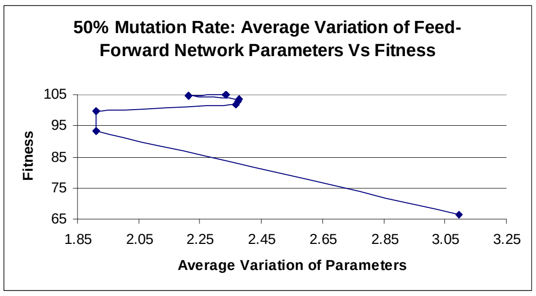
Figure 6.25 Average Performance of Feed-Forward versus Recurrent Networks

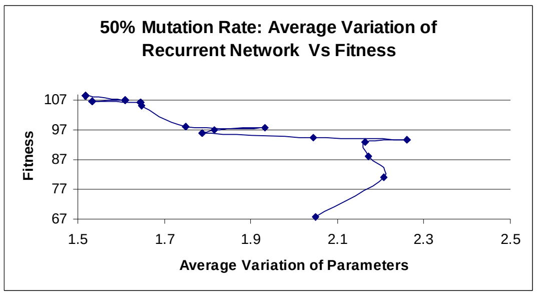
Figure 6.25 shows the average performance of the feed-forward network and the recurrent network. In each graph, the fitness of the network is plotted against the number of generations. Except for the graph in figure 6.25 (a), all the graphs show that the recurrent network performance is slightly better than the feed-forward network. However, the difference in performance is not significant, and anyway, the aim of the investigation was to observe the variations of parameter values and their outcome (fitness). These results are shown below in figure 6.26.



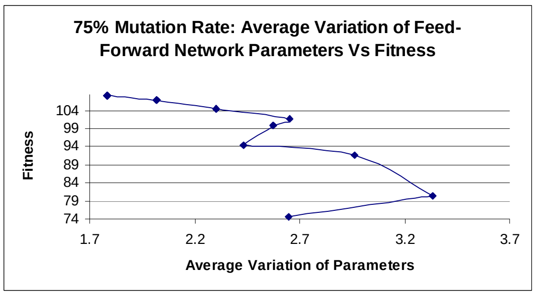


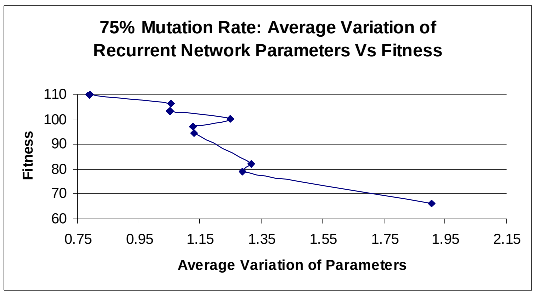
(a) (b)



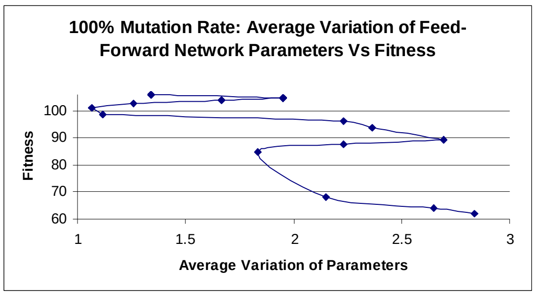


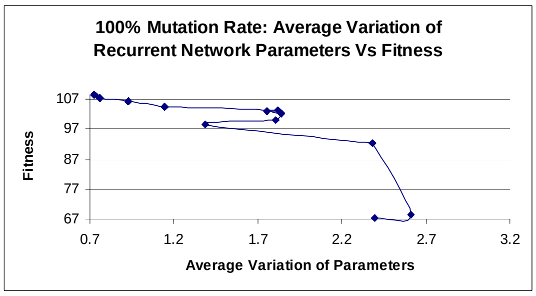
(c) (d)





(e) (f)





(g) (h)

Figure 6.26 Average Variation of Parameters versus Fitness

Each graph shown in figure 6.26 shows the average fitness of the networks plotted against their average variation of parameters. The results from the feed-forward network (refer to figure 6.26 (a), (c), (e) and (g)) showed that a mutation rate of 25% and 50% was causing the fitness to make big jumps and mutation rates of 75% and 100% was causing the fitness to increase by small increments. For example, consider the result of 25% mutation in a feed-forward network (refer to figure 6.26 (a)). The fitness has jumped from approximately 75 to 90 for the average variation of parameters from approximately 2.5 to 3.25. Now consider the result of 100% mutation rate in a feed-forward network (refer to figure 6.26 (g)). For approximately the same average variation of parameters (2 to 3), the fitness has increased by a small amount – from approximately 61 to 69.

The big jumps in fitness, in a real-world situation, could cause a problem in certain circumstances, for example in the presence of unexpected obstacles. Imagine such a network (shown in 25% mutation example) is controlling a robot. If that robot is climbing a rocky-steep-hill where it has to be careful about falling or slipping, an algorithm causing the robot to make big steps may cause it to not survive. The reason for these big jumps was investigated and is explained in the section 6.2.2.6.

The results from the recurrent network (refer to figure 6.26 (b), (d), (f) and (h)) showed a good performance compared to the feed-forward network (a good performance because of the increase in fitness by small values). This is due to the weights of the recurrent network; the network has two weights for each neuron which means that it is easier for the Net value to overcome the threshold; however even then a small variation in ON and OFF times can effect the output. Thus, the network showed a better performance than the feed-forward network. At this stage it was not clear whether the recurrent network could operate efficiently in real-time because the network was not tested with a real-time EA. Hence, both networks were used for further investigation. The maximum fitness achieved from different mutation rates were compared to learn more about the network's behaviour; this is shown in the next section.

6.2.2.5 Comparison of Mutation Rate

Figure 6.27 shows the results of a comparison of different mutation rates applied to the feed-forward and recurrent networks. The graph shows the maximum average fitness of the networks plotted against the experiment number (each number representing a different mutation rate). The first experiment started with a mutation rate of 12.5% and this was then incremented by 12.5% in each experiment (from left to right in the graph below – Number of Experiments axis). The graph shows that, in both the network performances, the maximum average fitness increases as mutation rate increases from 12.5% to 75%; then, as the mutation rate increases further, the average fitness decreases.

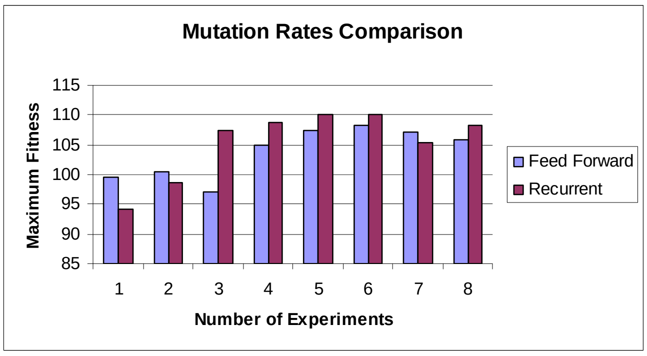


Figure 6.27 Mutation Rates Comparison

It was found that above 75%, when most or all of the parameters are mutated, performance decreases. This was investigated further and explained in section 6.2.2.6.

Summary of Investigation – including section 6.2.2.4

1. The results of feed-forward network with low mutation rates showed large jumps in fitness
2. Higher mutation rates performed poorly compared with low mutation rates

All these issues were investigated and led to the modification of the neuron model to make it more effective in real-time operation. This is explained in the next section.

6.2.2.6 Modifying the Neuron Model to be Effective in Real-Time

In this section the reason for the previously described behaviours of the networks are described along with an explanation of the modified neuron model.

1. Cause of Large Increments in Fitness

It was found from further inspection that the cause of the big jumps in fitness was due to the presence of the threshold parameter. For example, consider the result of a 50% mutation rate in a feed-forward network (refer to figure 6.26 (c)). The fitness has jumped from approximately 65 to 95. The total number of parameters is eight and with a 50% mutation rate; four parameters are mutated in each generation. Out of these four parameters, the chance of a weight being mutated is the same as a threshold and timing parameters (T_{ON} and T_{OFF}) mutated. This means that if the weights, thresholds and timing parameters are mutated at the same time, there is a good chance of the Net value failing to overcome the threshold, until at one point, the Net may overcome the threshold and cause the fitness to increase drastically. The following example is used to explain this scenario further.

1. $W_{21}(g) = 2$, $\theta_1(g) = 2.5$, $T_{ON} = 5$, $T_{OFF} = 5$ and $In(g) = 1$
2. $Net(g) = In(g) \times W_{21}(g) = 1 \times 2 = 2 < \theta_1(g) = 2.5$ – Net is less than the threshold $\theta_1(g)$ and hence the neuron will not fire.

Continued in Next Page

3) In the next generation (g+1), if the weight, threshold, T_{ON} and T_{OFF} are mutated, then the new values are as follows
4) $W_{21}(g+1) = 3$, $\theta_1(g+1) = 3.5$, $T_{ON}(g+1)= 7$ and $T_{OFF}(g+1)= 7$
5) Net = $1 \times 3 = 3 < \theta_1(g+1) = 3.5$ – Net is still less than the threshold $\theta_1(g+1)$ and hence neuron will not fire
6) In the next generation (g+2), if the weight, threshold, T_{ON} and T_{OFF} are mutated, then the new values are as follows
7) $W_{21}(g+2) = 4$, $\theta_1(g+2) = 3.75$, $T_{ON}(g+2)= 8$ and $T_{OFF}(g+2)= 8$
8) Net = $1 \times 4 = 4 > \theta_1(g+2) = 3.75$ – Net has overcome the threshold and as the T_{ON} and T_{OFF} has increased from 5 to 8 and the fitness would increase drastically.

Thus, the threshold can act as a barrier and cause the network to make big jumps in fitness. It was therefore decided to remove the threshold parameter from the neuron model.

2. Higher Mutation Rates

The weights and the threshold take part in the activation process of the neuron (refer to activation function, equation 6.3). The ON/OFF time decides the stride length of the robot, which in turn impacts on its fitness (refer to distance equation 6.2 and fitness function, equation 6.4). As the three parameters of the neuron; weights, threshold and ON/OFF time are dependent of each other, applying higher mutation rates may result in poorer performance. For example, when a 100% mutation rate is applied, the weights and the thresholds of the neurons are both mutated and this may cause both the weights and the thresholds to increase or decrease proportionally. The following example is used to show one possible scenario. This example uses the feed-forward network topology, as shown in figure 6.19. Only neuron 1 (N1) is shown here for demonstration.

N1:

$In_1 = 1$ – Input to the neuron 1

$W_{21} = 2$ – Weight of the neuron 1

$\theta_1 = 3$ – Threshold of the neuron 1

Generation 1: Net = $In_1 \times W_{12} = 1 \times 2 = 3$

Net $< \theta_1$

Hence neuron is not activated

Generation 2: After 100% mutation is applied, if new $W_{12} = 3$ and is $\theta_1 = 4$

Net = 1 x 3 = 3

Net < θ_1

Net is still less than threshold and hence neuron is not activated. As result it shows a poorer performance.

Thus it is possible for the Net not being able overcome the threshold. This stops the network from improving its fitness. Thus, higher mutation rates cause the network to perform worse than is the case when smaller mutation rates are applied.

To allow the changes in one of the entities to affect the other, it was found that the value Net can be set as ON-time. Hence, by varying the weights, the ON-time of the neuron can be varied. To get the OFF-time of the neuron, a new constant parameter was introduced called the T_3 – the sum of ON and OFF-time. After calculating the Net (ON-time), the OFF-time was calculated by subtracting the Net value from the T_3. Hence, with the threshold being removed, the new modified model had the following parameters. In this, the weight is the only evolvable parameter.

T_3 – Sum of ON and OFF Times ($T_{ON} + T_{OFF}$, Constant value)
W – Weight (The number of weights depends on the number of neurons connected)

The neuron activation function is given in equation 6.5.

$$\begin{aligned}
&if\,(Net_i(t) > 0) \quad for \quad 0 < t < (T_{ONi} = Net_i) &&\Rightarrow Out_i(t) = +1 \\
&where \quad Net_j = \sum_{i=1}^{n} In_i \times W_i && \\
&and \qquad\qquad for \quad 0 < t < (T_{OFFi} = (T_3 - Net_i)) &&\Rightarrow Out_i(t) = -1 \\
&if\,(Net_i(t) \le 0) \quad for \quad 0 < t < \infty &&\Rightarrow Out_i(t) = -1
\end{aligned} \tag{6.5}$$

Where *Net_j* is the sum of product of input and weight – Neuron potential of j^{th} neuron

Out_j is the output of j^{th} neuron

In_i is the input to the j^{th} neuron

W_i is the weight of the j^{th} neuron

T_{ONj} is the On time of the j^{th} neuron

T_{OFFj} is the Off time of the j^{th} neuron

T_3 is the sum of T_{ONi} and T_{OFFi}

At this stage of the project, enough information had been gathered to code the Initial Implementation of the RTEA (II-RTEA) which could then be structured to control the robot in real-time.

In the next section the operation of II-RTEA is explained with simulated results comparing the performances of the feed-forward network and the recurrent network.

6.2.2.7 II-RTEA operation of the Networks

As discussed in chapter 4, the II-RTEA had, of necessity, no recombination operator and used only a mutation operator to evolve the network parameters. It operated on a single string of parameters in the network. The single string of parameters for the feed-forward network is shown in figure 6.28(a) and for the recurrent network, shown in figure 6.28(b). For recurrent network topology, refer to figure 6.11 and for feed-forward network topology, refer to figure 6.18.

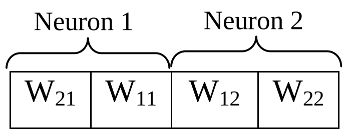


Figure 6.28 (a) Recurrent Network String of Parameters

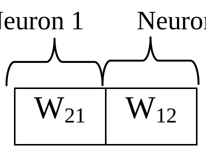


Figure 6.28 (b) Feed-Forward Network String of Parameters

The simulation method of the II-RTEA is shown in figure 6.29. The simulation starts by initializing random values to the single string of parameters using a Uniform Random Number Distribution (refer to chapter 4, section 4.2.3).

The network was simulated using the equation 6.5. Then the robot was simulated and the fitness evaluated using equation 6.4. If the network error increases or the fitness decreases (compared with the previous fitness), the changes are discarded; if fitness increases the changes are accepted. The steps above are repeated for the specified number of generations. Using this simulation method, the feed-forward and recurrent networks were simulated and the results are shown below.

A similar experimental setup to that used in section 6.2.2.4 was used here, except that the training algorithm is II-RTEA (no population and no recombination) and the evolvable parameters are different as shown below. The results are also shown figure 6.30.

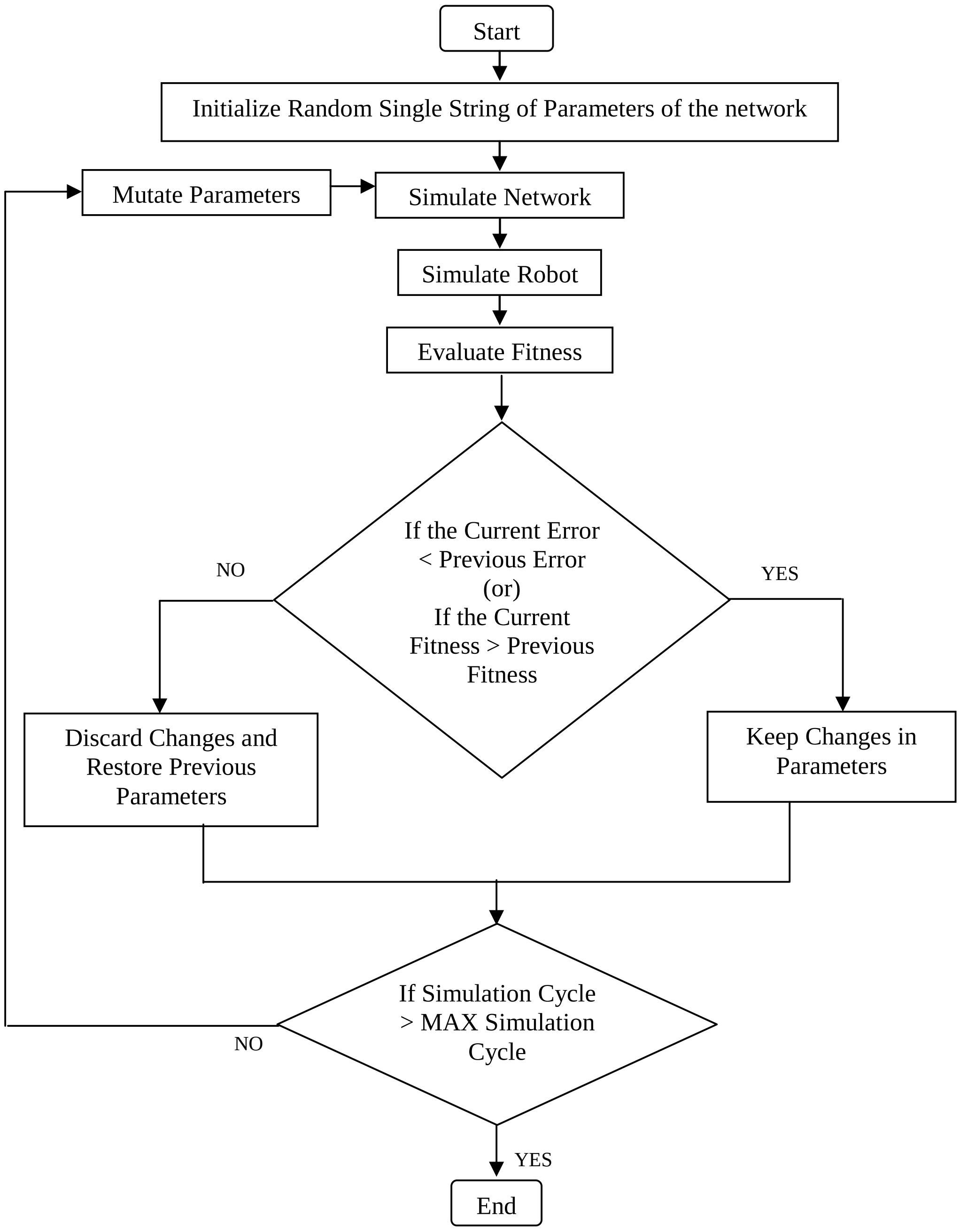
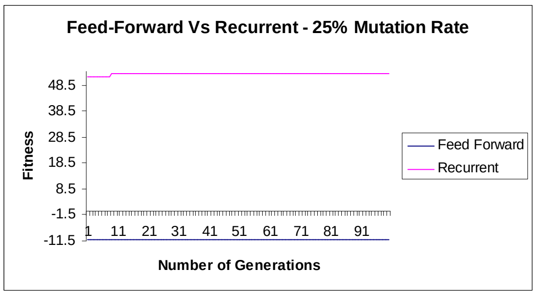
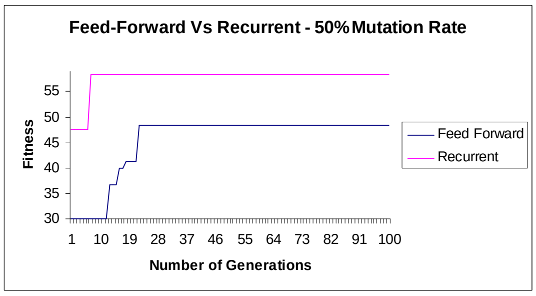

Start
Initialize Random Single String of Parameters of the network
Mutate Parameters
Simulate Network
Simulate Robot
Evaluate Fitness
If the Current Error < Previous Error (or) If the Current Fitness > Previous Fitness
NO
YES
Discard Changes and Restore Previous Parameters
Keep Changes in Parameters
If Simulation Cycle > MAX Simulation Cycle
NO
YES
End

Figure 6.29 II-RTEA Simulation Method

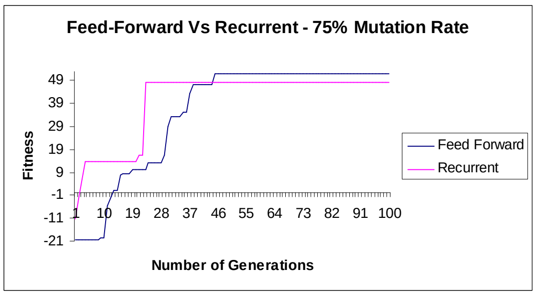
Feed Forward Network	**Recurrent Network**
1. Evolvable parameters:	1. Evolvable Parameters
Neuron 1 –N1: W_{21}	Neuron 1 – N1: W_{11} and W_{21}
Neuron 2 – N2: W_{12}	Neuron 2 – N2: W_{12} and W_{22}



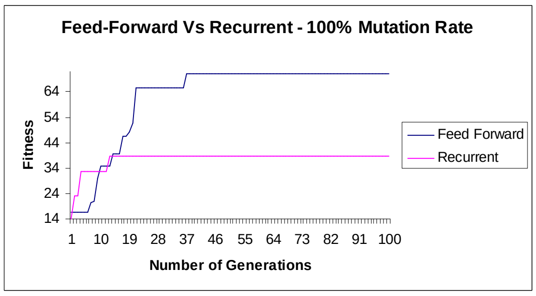
(a)



(b)



(c)

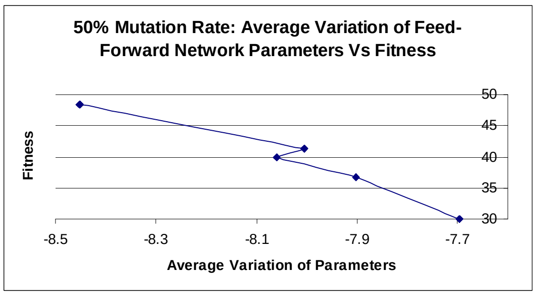


(d)

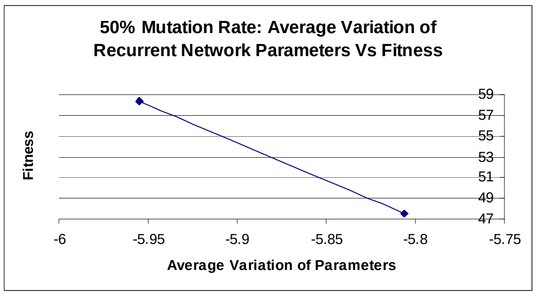
Figure 6.30 Feed-Forward Vs Recurrent – II-RTEA Performance

The graphs in figure 6.30, show the average performance of the feed-forward network and recurrent networks, trained using II-RTEA. In each graph, the fitness of the network is plotted against the number of generations. The result with a mutation rate of 25% in the feed-forward network (refer to figure 6.30 (a)) shows no improvement in fitness, because the mutation rate is very low. The total number of parameters in the network is two. A mutation rate of 25% is not possible in the feed-forward network. The result with the recurrent network (refer to figure 6.30 (a)) shows an increase in fitness only once. This is because the simulation program rounds up values higher than 0.5 to 1, and the total number of parameters in the network is three. The mutation rate of 25% acts on 0.75 neurons and so the program rounds up the value (0.75 to 1). However, it was learned from these results that the mutation rate is critical for the II-RTEA to operate correctly.

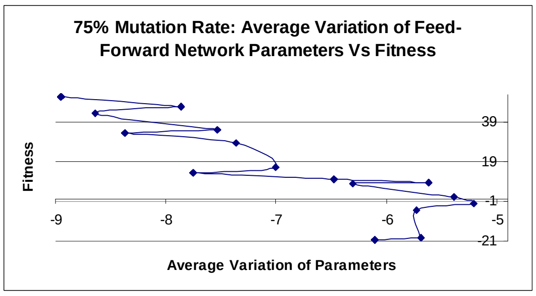
From the above results, it was noticed that the feed-forward network performed better than the recurrent network with mutation rates of 75% and 100%. Also it was noticed that the recurrent network showed a big jump in fitness – this can be noticed particularly in the figure 6.30 (c), between generation 19 and 28. The disadvantage of this big jump was explained with an example in section 6.2.2.6 and it applies here as well. To investigate this issue further, the average variation of parameters and their fitness was compared for both the networks. The graphs below show these comparisons. Note that the results with a mutation rate 25% were not compared because they did not show any increase in fitness (refer to figure 6.30 (a)). The rest of the results are shown in figure 6.31.



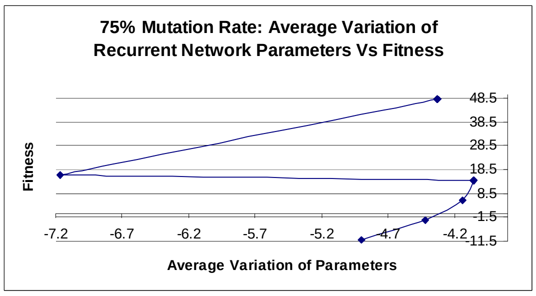
(a)



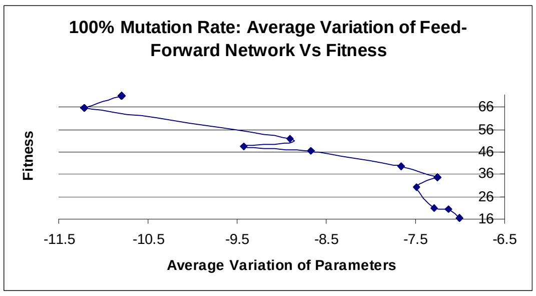
(b)



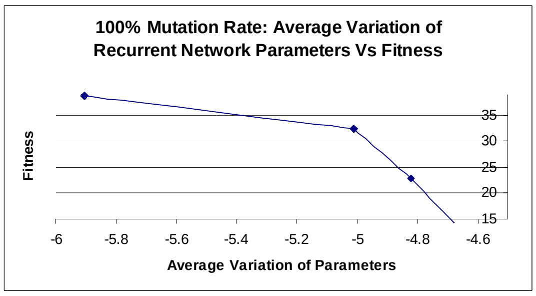
(c)



(d)



(e)



(f)

Figure 6.31 Average Variation of Parameters versus Fitness of Feed-Forward and Recurrent Networks

The results in figure 6.31 (a), (c) and (e) show that the feed-forward network has shown a gradual increase in fitness. The results in figure 6.31 (b), (d) and (f), show that the recurrent network has shown a large jump in fitness. This can be particularly noticed in figure 6.31 (d), where the fitness has jumped from approximately 10 to 50. These big jumps may be due to the feedback in the recurrent network. The following example illustrates this issue.

1. $W_{11}(g) = -4$, $W_{21}(g) = -4$ and $In(g) = 1$
2. $Net(g) = (In(g) \times W_{11}(g)) + (In(g) \times W_{21}(g)) = (1 \times -4)+(1 \times -4) = -8 < 0$ – Net is less than the zero and hence the neuron will not fire.
3. In the next generation (g+1), if the weights are mutated, then the new values are as follows
4. $W_{11}(g+1) = 4$, $W_{21}(g+1) = -4$. **Note:** W_{11} has changed
5. $Net = (1 \times 4)+(1 \times -4) = 0 = 0$ – Net is equal to zero and hence neuron will not fire
6. In the next generation (g+2), if the weights are mutated, then the new values are as follows
7. $W_{11}(g+2) = 8$, $W_{21}(g+1) = 4$ **Note:** W_{11} and W_{21} have changed
8. $Net = (1 \times 8)+(1 \times 4) = 12 > 0 \Rightarrow T_{ON} = Net$ – which results in huge increase in fitness

Thus, it was found that the recurrent connection sometimes acts like the threshold discussed in section 6.2.2.6. It stops the neuron from performing in a useful way and also occasionally causes the network to make big jumps in fitness. If the neurons did not have self-recurrent connections; the situation would have been as illustrated below.

1. $W_{21}(g) = -4$ and $In(g) = 1$
2. $Net(g) = (In(g) \times W_{21}(g) = (1 \times -4) = -4< 0$ – Net is less than the zero and hence the neuron will not fire.
3. In the next generation (g+1), if the weights are mutated, then the new values are as follows
4. $W_{21}(g+1) = -4$.
5. $Net = (1 \times -4) = -4 < 0$ – Net is less than zero and hence neuron will not fire
6. In the next generation (g+2), if the weights are mutated, then the new values are as follows

7. W21(g+2) = 4 Note: W21 has changed
8. Net = (1 x 4) = 4 > 0 $\Rightarrow$ TON = Net – which results in relatively making smaller increase in fitness compared to what happed with recurrent network

Thus, the feed-forward network performed better than the recurrent network when operated by II-RTEA in real-time.

6.3 Lessons Learned from Biped

The important points learned from the bipeds are as follows:

1. **Independent Parameters**

   The initial investigation into the neuron model showed that, if the neuron has different parameters such as weights, thresholds and timing parameters, the evolve-ability becomes inconsistent. This inconsistency is due to mutation – mutating different parameters in every generation can make the network evolve in any direction, which results in inconsistency. This was learned from the results shown in sections 6.2.2.4 and 6.2.2.5 and led to the modification of the neuron model, making the Net of the neuron set T_{ON} (and T_{OFF} as (T_3-T_{ON}), where T_3 is the sum of T_{ON} and T_{OFF}).

2. **Thresholds**

   The threshold may be a barrier to the neuron behaving in a useful way. This was learned from the results shown in section 6.2.2.7 and the threshold parameter was removed from the neuron model.

3. **Self-Recurrent Connections**

   Sometimes, due to recurrent connections, the network may show big jumps in fitness where a gradual increase is wanted. The recurrent connection can hold back any increase in fitness for a certain number of iterations and then in one iteration the result may be an unexpectedly large change in output. This was observed from the results shown in section 6.2.2.8. Hence, it was learned that the network should be simple and efficient to achieve good control in real-time. The results showed that a simple feed-forward network without a threshold performs well in real-time.

6.4 Summary

This chapter started by explaining the development of a biped simulator. Then the different alternative neuron models and network topologies were discussed. The outcome of this investigation was that the time-dependent neuron model with Net serving as on-time and no threshold performed best and so it was selected for further investigation. It was also found that a simple feed-forward network topology performed better than recurrent networks in real-time. The initial implementation of the algorithm's (II-RTEA) function was also explained in this chapter. The next chapter starts by explaining how the lessons learned from the biped simulations were extended to the development of a quadruped simulator and controlling network.

Chapter 7

From Bipeds to Quadrupeds

7.1 Introduction

This chapter explains how the lessons learned from the biped were used to develop a quadruped simulator and neural network. It starts with the development of the simulator and then describes the fitness function used. Finally, the neural network development for the quadruped is explained.

7.2 Development of Simulator from Biped to Quadruped

This section explains the development of the robot simulator from biped to quadruped. The quadruped robot used as a basis for simulation was the "Lynx Motion", servo-controlled four-legged robot. The real robot, on which the simulation is based, is shown in figure 7.1.

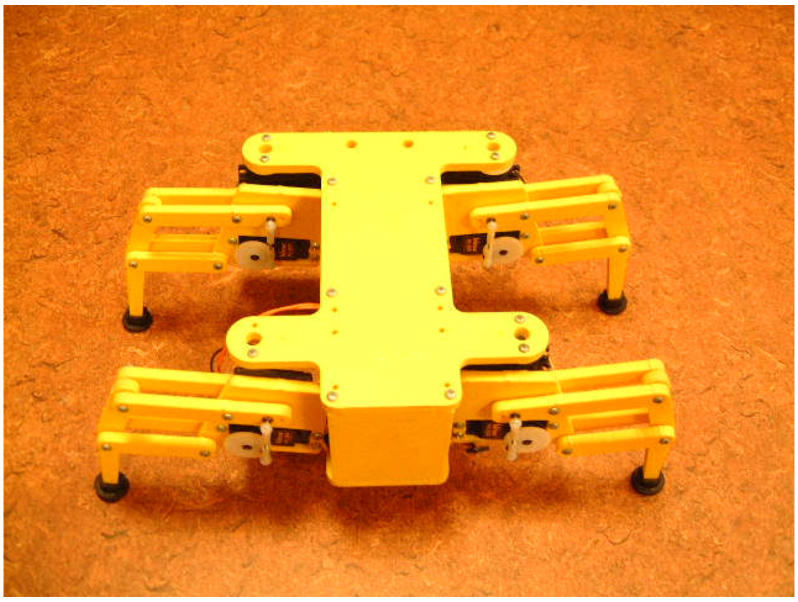


Figure 7.1 Quadruped Robot

Figure 7.2 shows a simple schematic diagram of the robot. As may be seen, the robot has active degrees of freedom in the hip joints and passive degrees of freedom in the knee joints.

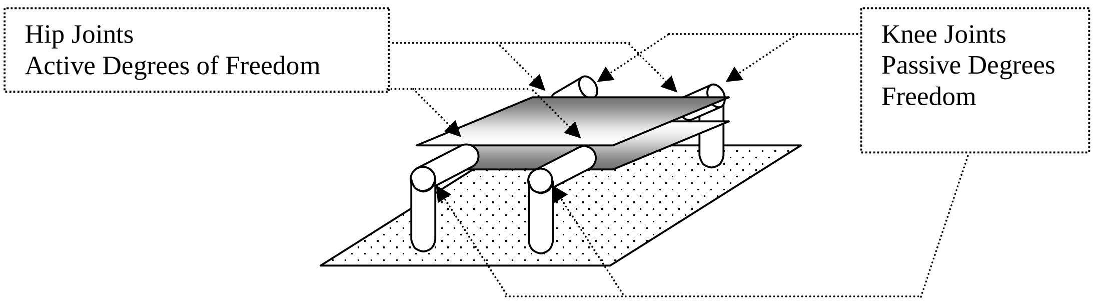
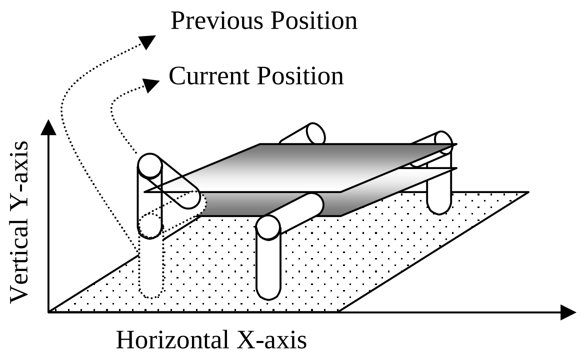


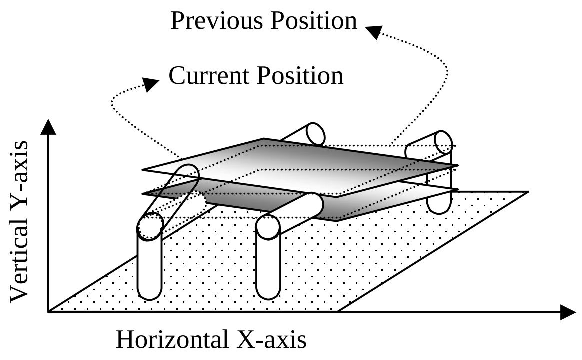
Figure 7.2 Schematic Diagram of Quadruped Robot

The hip joints have two active degrees of freedom – they can make vertical and horizontal movements, as shown in figure 7.3 (a), (b) and (c). Note that as shown in figure 7.3 (b), a vertical-downward movement causes the body of the robot to rise as the leg pushes itself up against the floor. These two degrees of freedom are driven by two separate servos.



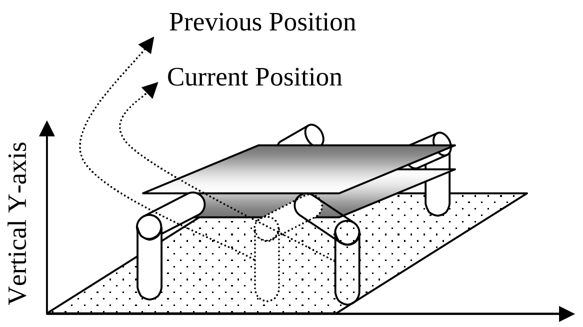
Vertical-Up Movement
In this example (left figure), the back left leg is shown to be performing a vertical-up movement. The action is a result of servomotor's **anti-clockwise** rotation. This applies to all the four legs of the robot.

(a) Vertical-Up Movement



Vertical-Down Movement
In this example (left figure), the back left leg is shown to be performing a vertical-down movement. The action is a result of servomotor's **clockwise** rotation. When the servo rotates clockwise the leg pushes itself against the floor and as a result, the body tend to move upwards, depending on the other positions of the legs. This applies to all the four legs of the robot.

(b) Vertical-Down Movement



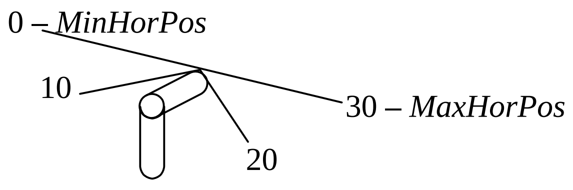
Horizontal Movement
In this example (left figure), the back left leg is shown to be performing a horizontal-forward movement. The action is a result of servomotor's **clockwise** rotation. Similarly if the servo rotates **anti-clockwise**, the leg moves backwards. This applies to all the four legs of the robot.

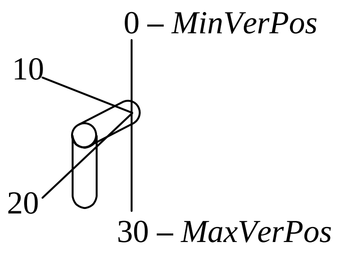
(c) Horizontal movement

Figure 7.3 Two Degrees of Freedom Hip Joint

Similarly to the biped's actuator, the quadruped actuators move anticlockwise on receiving a positive input pulse and clockwise on receiving a negative pulse. The quadruped's leg movements were restricted to the positions shown in figure 7.4 (a) and (b).

The dynamics of movement (the relationship between control inputs and the leg movements) are the same as those previously described for the biped system. The robot moves forward by lifting its leg (vertical move), moving the leg forward (horizontal move), dropping the leg to the floor (vertical move) and finally moving the leg backwards (horizontal move). To perform a specific gait (like walk or gallop), all the leg movements have to be timed correctly. For example, the snap-shots shown below, in figure 7.5, are a simulation of a quadruped walk (the snap-shots do not give a good resolution; however, the leg movements can be seen on close inspection). Each time-frame shows a side-view and a three-dimensional view of the robot along with a description of the leg movement states.





(a) Horizontal Movement Restriction (b) Vertical Movement Restriction

Where *MinHorPos* = Minimum Horizontal Position

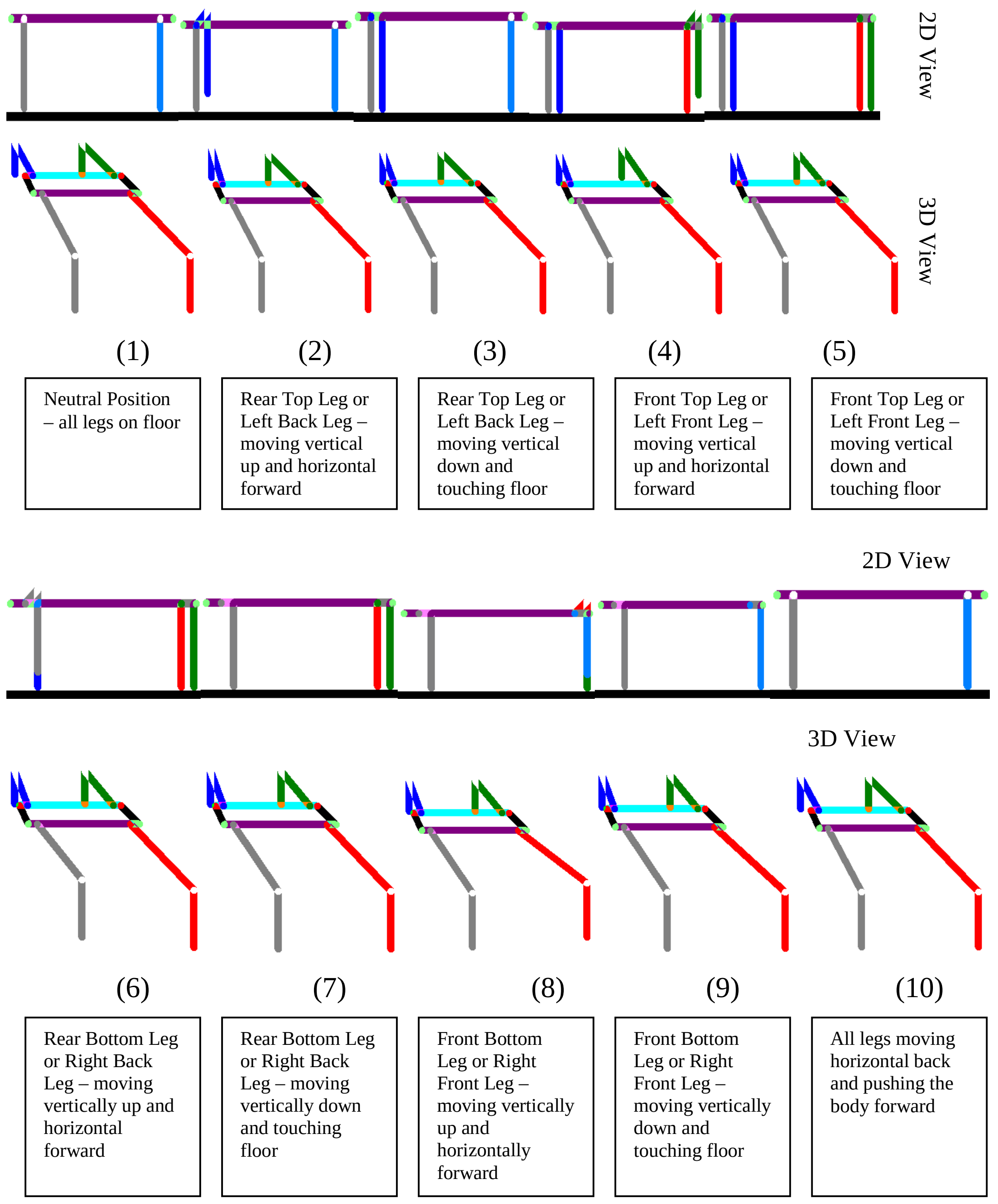
MaxHorPos = Maximum Horizontal Position

MinVerPos = Minimum Vertical Position

MaxVerPos = Maximum Vertical Position

Figure 7.4 Leg Movement Restriction

It can be seen that the different legs move at different times and show more complexity than the biped's motion.



2D View
3D View
(1)
Neutral Position – all legs on floor
(2)
Rear Top Leg or Left Back Leg – moving vertical up and horizontal forward
(3)
Rear Top Leg or Left Back Leg – moving vertical down and touching floor
(4)
Front Top Leg or Left Front Leg – moving vertical up and horizontal forward
(5)
Front Top Leg or Left Front Leg – moving vertical down and touching floor
2D View
3D View
(6)
Rear Bottom Leg or Right Back Leg – moving vertically up and horizontal forward
(7)
Rear Bottom Leg or Right Back Leg – moving vertically down and touching floor
(8)
Front Bottom Leg or Right Front Leg – moving vertically up and horizontally forward
(9)
Front Bottom Leg or Right Front Leg – moving vertically down and touching floor
(10)
All legs moving horizontal back and pushing the body forward

Figure 7.5 Quadruped Robot Simulation – Walking Gait

The parameters used in the robot simulation are as follows. They apply to every leg of the robot (including the equations from 7.1 to 7.3).

Horizontal Movement	Vertical Movement
Leg_{HorAct} – Leg's Horizontal Actuator	Leg_{VerAct} – Leg's Vertical Actuator
Leg_{HorPos} – Leg's Horizontal Position	Leg_{VerPos} – Leg's Vertical Position

Apart from the horizontal and vertical parameters, every foot has a simulated touch sensor. This simply senses if the foot is touching the flat surface or not. It receives a signal-value of '+1' if it is touching the floor and a signal-value of '-1' if it is not. The function of the sensor is given in equation 7.1.

$$\begin{aligned} &if\,(Leg_{FootSensor} = +1) \Rightarrow True,\ \ leg\ \ touching\ \ floor \\ &if\,(Leg_{FootSensor} = -1) \Rightarrow False,\ \ leg\ \ Not\ \ touching\ \ floor \end{aligned} \tag{7.1}$$

Where $Leg_{FootSensor}$ is the Foot Sensor

The relationship between the leg's actuator and the leg's position at a given time 't' is governed by equation 7.2.

$$\begin{aligned} &if\,(Leg_{HorAct}(t) = +1) \Rightarrow Leg_{HorPos}(t) = Leg_{HorPos}(t) + 1 \\ &if\,(Leg_{HorAct}(t) = -1) \Rightarrow Leg_{HorPos}(t) = Leg_{HorPos}(t) - 1 \\ \\ &if\,(Leg_{VerAct}(t) = +1) \Rightarrow Leg_{VerPos}(t) = Leg_{VerPos}(t) + 1 \\ &if\,(Leg_{VerAct}(t) = -1) \Rightarrow Leg_{VerPos}(t) = Leg_{VerPos}(t) - 1 \end{aligned} \tag{7.2}$$

Where Leg_{HorAct} is the Leg's Horizontal Actuator

Leg_{HorPos} is the Leg's Horizontal Position

Leg_{VerAct} is the Leg's Vertical Actuator

Leg_{VerPos} is the Leg's Vertical Position

The distance moved by the robot and the corresponding actuator's function is given by equation 7.3.

$$\begin{aligned}&if\,(Leg_{HorAct}(t) = -1 \;\; and \;\; 10 < Leg_{HorPos}(t) < 20 \;\; and \;\; Leg_{FootSensor}(t) = True)\\ &\Rightarrow d_t = d_t + 1\end{aligned} \tag{7.3}$$

Where d_t is the Distance moved at time ‘t’

Leg_{HorAct} is the Leg’s Horizontal Actuator

Leg_{HorPos} is the Leg’s Horizontal Position

$Leg_{FootSensor}$ is the Foot Sensor

7.3 Development of Fitness-Function from Biped to Quadruped

The accuracy of the fitness-function plays an important role in evolving a system. The more precise the fitness-function, the better the evolutionary result is. In this project, to evolve and control quadruped networks, the fitness function was carefully structured. This section explains the fitness-function used in this project.

The quadruped model used in the project has no complex sensors like sonar, cameras, infrared sensors, etc. It must work using only its available sensor information. Three fitness parameters can be used to measure robot’s performance as shown in equation 7.4.

$$fitness = (S \times R_S\,\%) + (D \times R_D\,\%) + (F \times R_F\,\%) \tag{7.4}$$

Where S is the Stability of the robot

R_S is constant expressing the importance of stability

D is the Distance covered by the robot

R_D is constant expressing the importance of Distance

F is the Fuel consumption

R_F is constant expressing the importance of Fuel

R_S, R_D and R_F are experimental parameters, set up by the user and explained in the next chapter. Each of the three parameters (S, D and F), was calculated over a certain number of Time-Steps by simulating the robot. The three fitness parameters and their method of measurement are explained below.

1. Stability Parameter

Stability is a measure of how likely the robot is to fall over. To measure stability, the inclination of the robot and the number of feet it has on the ground over the specified number of Time-Steps is calculated. The expression used is shown in equation 7.5.

$$S = \left(\frac{L_T}{T}\times 100\right) + \left(\frac{L_D}{T}\times 100\right) + \left(\frac{L_S}{T}\times 100\right) + \left(\frac{L_{F/B}}{T}\times 100\right) - \left(\frac{L_E}{T}\times 100\right) \qquad (7.5)$$

Where S is the Stability

T is the number of Time-Steps to the simulation (typically zero)

L_T is Three Legs on the ground

L_D is the Diagonal Legs on the ground

L_S is the Side legs on the ground

$L_{F/B}$ is the Front two legs or Back two legs on the ground

L_E is the Error made by the leg – body inclination

These parameters represent the different possibilities of leg positions, related to stability. In each Time-Step, the numbers of legs on the ground is counted and the stability parameters (shown in equation 7.5) are altered as shown below.

$$\begin{aligned}
&if\,(LegOnGround = 3) \Rightarrow L_T = L_T + 1 \\
&if\,(LegOnGround = 2)\quad and \\
&\qquad if\,(DiagonalLegsOnGround = true) \Rightarrow L_D = L_D + 1 \\
&\qquad if\,(SideLegsOnGround = true) \Rightarrow L_S = L_S + 1 \\
&\qquad if\,(Front/BackLegsOnGround = true) \Rightarrow L_{F/B} = L_{F/B} + 1 \\
&if\,(RobotBodyInclined) \Rightarrow L_E = L_E + 1
\end{aligned} \qquad (7.6)$$

Where L_T is Three Legs on the ground

L_D is the Diagonal Legs on the ground

L_S is the Side legs on the ground

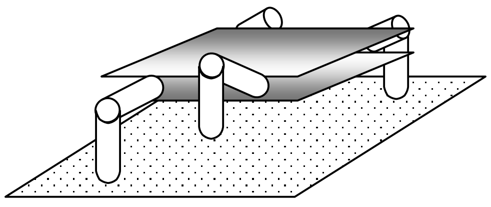
$L_{F/B}$ is the Front two legs or Back two legs on the ground

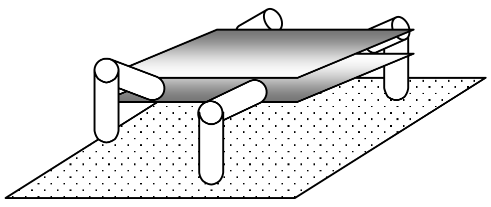
L_E is the Error made by the leg – body inclination

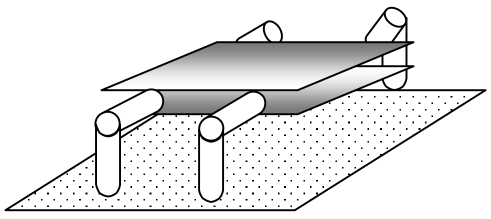
An example of the measurement of each stability parameter is given below with suitable diagrams.

a. *Three legs on the ground*

In the specified Time-Steps, the robot may be in any of the positions shown in figure 7.6. Each is a case were the stability parameter L_T is incremented by one unit.


(a) Front right leg off the ground


(b) Back right leg off the ground


(c) Front left leg off the ground

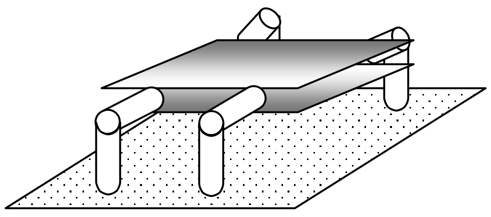

(d) Back left leg off the ground

Figure 7.6 Three legs on the ground

b. *Diagonal legs the ground*

The cases of parameter L_D being incremented by one unit are shown in figure 7.7.

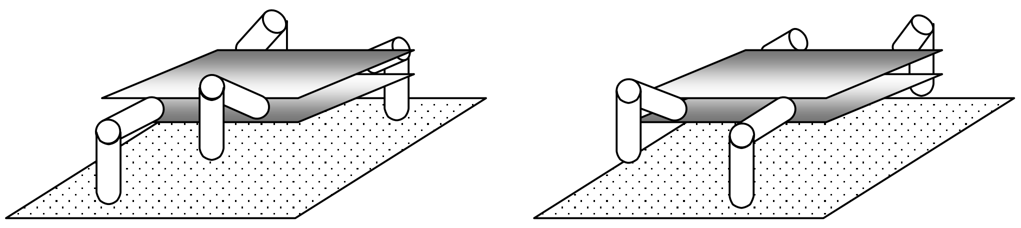


Figure 7.7 Two legs on the ground – Diagonal Legs

c. *Side Legs on the ground*

The scenarios where parameter L_S is incremented by one unit are shown in figure 7.8.

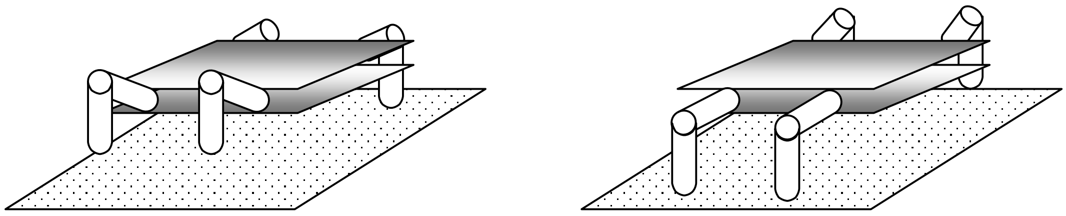


Figure 7.8 Two legs on the ground – Side Legs

d. *Front two or Back two legs on the ground*

Finally, stability parameter $L_{F/B}$ is incremented by one unit on the cases shown in figure 7.9.

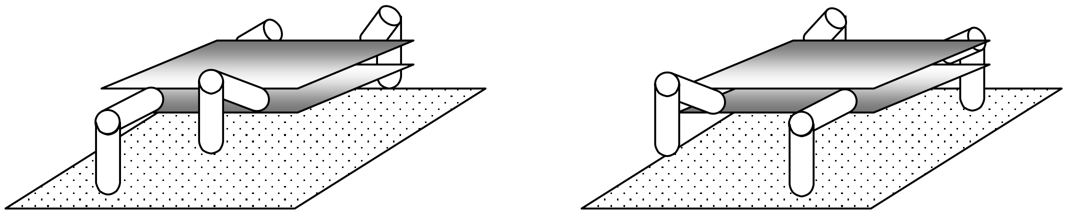


Figure 7.9 Two legs on the ground – Front/Back legs

e. *Leg Error – Body inclination*

Body inclination can be caused by an irregular vertical-down move. One such circumstance is shown in figure 7.10. In such scenarios, L_E is incremented by one unit.

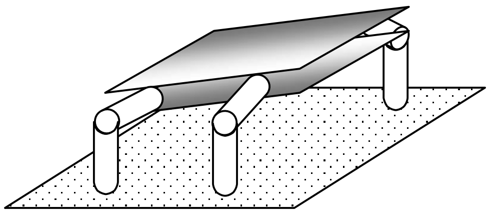


Figure 7.10 Leg Error – Body Inclination

2. Distance Parameter

The distance covered by the robot over a specified period is given by equation 7.7.

$$D = \left(\frac{D_{FL}}{T} \times 100\right) + \left(\frac{D_{BL}}{T} \times 100\right) - \left(\frac{D_E}{T} \times 100\right) \qquad (7.7)$$

Where *D* is the Distance travelled by the robot

T is the number of Time-Steps

D_{FL} is the Distance covered by using the front legs

D_{BL} is the Distance covered by using the back legs

D_E is the Error in covering the distance

The method of measuring each distance is shown below.

$$\begin{aligned}
&if\,(Leg_{HorPos}(t-1) > Leg_{HorPos}(t) \quad and \quad (Leg_{FootSenor}(t) = True)) \quad and \\
&\qquad if\,(LegUsedToMove = FrontLeg) \Rightarrow D_{FL} = D_{FL} + 1 \\
&\qquad if\,(LegUsedToMove = BackLeg) \Rightarrow D_{BL} = D_{BL} + 2
\end{aligned} \tag{7.8}$$

$$if\,(Leg_{HorPos}(t-1) < Leg_{HorPos}(t) \quad and \quad (Leg_{FootSenor}(t) = True)) \Rightarrow D_E = D_E + 1$$

Where D_{FL} is the Distance covered by using the front legs

D_{BL} is the Distance covered by using the back legs

D_E is the Error in covering the distance

The above method applies to each leg of the robot. The robot may use either or both of its front legs to move its body and similarly it can use either or both of its back legs. Hence, two separate parameters D_{FL} and D_{BF}, as shown in equation 7.8, were used to measure the distance covered. It was assumed that the back legs are power legs and hence the D_{BF} parameter was incremented by two units. It was also assumed that the front legs were not as powerful as the back legs and hence D_{FL} parameters are incremented by one unit. If the leg is moving backwards with the foot on the floor, this causes friction between the foot and the floor and could cause damage. This was penalised using the parameter D_E. Every time the leg makes such an error, D_E is incremented and a percentage of error is subtracted from the overall distance covered as shown in equation 7.7.

3. **Fuel Parameter**

Unlike the stability and distance measurement which used dedicated parameters, the fuel consumption of the robot is decremented from its given initial fuel quantity. The typical value assigned for the quadruped in this project is 1000 units. The units of fuel decremented depends on the nature of the action at a given Time-Step –‘t’. The equation used is given below (equation 7.9).

$$F = \left(\frac{F_U}{F_I}\right) \times 100 \qquad (7.9)$$

Where F is the Fuel consumed

F_U is the Fuel Used

F_I is the Initial fuel

The method of calculation is given below. This explains how the F_U parameter is decremented according to the nature of the robot's action (penalties were chosen arbitrarily between 1 and 10).

	F_U Decrementing Condition	Robot's Action
1	*if((Leg$_{HorPos}$(t) > Max$_{HorPos}$) or (Leg$_{HorPos}$(t) < Min$_{HorPos}$) or (Leg$_{VerrPos}$(t) > Max$_{VerPos}$) or (Leg$_{VerPos}$(t) < Min$_{VerPos}$))* $\Rightarrow$***F$_U$ = F$_U$ – 10***	In a real robot, if the leg tries to move beyond its maximum limits, the servomotors may get damaged. In the simulation it was assumed that this consumed ten units of fuel and was thereby penalized.
2	*if(AllFourLegsAreOffTheGround)* $\Rightarrow$***F$_U$ = F$_U$ – 8***	A jump requires more energy. In simulation this was assumed to consume eight units of fuel.
3	*if(OneLegOnGround)* $\Rightarrow$***F$_U$ = F$_U$ – 6***	A robot balancing on one leg, expends energy. In simulation this was assumed to consume six units of fuel.
4	*if(SideLegsOnGround)* $\Rightarrow$***F$_U$ = F$_U$ – 5***	For a robot to balance itself using either of the side legs is also energy expensive. In simulation this was assumed to consume five units of fuel.
5	*if(Front/BackLegsOnGround)* $\Rightarrow$***F$_U$ = F$_U$ – 3***	Similarly, a robot to balance itself using its front or back legs. In simulation this was

		assumed to consume three units of fuel.
6	*if(DiagonalLegsOnGround)* $\Rightarrow \boldsymbol{F_U = F_U - 2}$	For a robot to balance itself using its two diagonal legs, in simulation, this was assumed to consume five units of fuel.
7	*if((Leg$_{HorPos}$(t-1) < Leg$_{HorPos}$(t)) and (Leg$_{FootSensor}$(t) = True))* $\Rightarrow \boldsymbol{F_U = F_U - 2}$	If the robot is brushing any of its legs on the floor, this may damage the system and in simulation this was assumed to consume two units of fuel (and thereby penalize this action).
8	*if(ThreeLegsOnGround)* $\Rightarrow \boldsymbol{F_U = F_U - 1}$	The robot is substantially stable if it has three legs on the floor (assuming it is on a stable surface). Hence, to have one leg in the air would not require as much as energy consumed as for other actions. In simulation this was assumed to consume only one unit of fuel.
9	*if(AnyNormalMovement)* $\Rightarrow \boldsymbol{F_U = F_U - 1}$	For any other normal action, the simulation assumed a consumption of one unit of fuel.

7.4 Development of Neural Network for Quadruped

7.4.1 Neuron Model

The investigation into the biped system showed that a time-dependent neuron, used in a simple feed-forward network, can effectively control robots with a RTEA. The investigation was next extended to developing a network suitable for quadruped control. As described in section 7.2, the legs of the robot not only have to move forward and backward, but also have to pause between actions. For example, in figure 7.5, in time-frame (2) and (3), the left-back leg has moved forward by performing vertical and horizontal motions, after which it has paused until time-frame (10). This is unlike the biped's continuous clockwise and anticlockwise action.

To accommodate this, a new time-property called "delay-time" or "pause-time" was introduced into the neuron model. This new property also came with a problem. The neuron model of the biped uses the Net value as the On-time and (On+Off)-time as a constant (refer to chapter 6, section 6.2.2.6). So, only one time-parameter can use Net. Introducing delay-time adds one more time-parameter variable to the list and hence the Net can no longer serve as the On-time – if Net is On-time, then what can be used for delay-time? It was decided therefore to reappraise to the previous model, where Net is used to simply trigger the neuron and the ON/OFF times are set by the EA. Although this model was not efficient in the biped, it was tested on the quadruped before ruling it out. Figure 7.11 shows the new time parameters and properties.

Neuron Parameters:
D_1 – Delay for ON time
T_{ON} – On time
D_2 – Delay for OFF time
T_{OFF} – Off time
Wij – Weight of neuron connected between ith and jth neuron

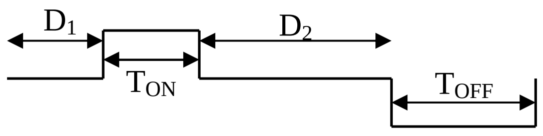


Figure 7.11 Neuron output

The activation function of the upgraded neuron model is shown below in equation 7.10.

$$\begin{aligned}
&if\,(Net_j(t)>0) \Rightarrow for \quad 0<t<D_{1j} \quad :Out_j(t)=0 \quad and\\
&\qquad\qquad for \quad 0<t<T_{ONj} \quad :Out_j(t)=+1 \quad and\\
&\qquad\qquad for \quad 0<t<D_{2j} \quad :Out_j(t)=0 \quad and\\
&\qquad\qquad for \quad 0<t<T_{OFFj} \quad :Out_j(t)=-1 \qquad\qquad (7.10)\\
&if\,(Net_j(t)<0) \Rightarrow for \quad 0<t<\infty \quad Out_j(t)=0\\
&\quad where \quad Net_j = \sum_{i=\,i^{th}\ neuron\ connected\ to\ j} In_i \times W_i
\end{aligned}$$

Where Net_j is the sum of the product of the inputs and weights

Out_j is the output of j^{th} neuron

In_i is the i^{th} input to the j^{th} neuron

W_i is the i^{th} weight of the j^{th} neuron

D_{1j} is the On-Delay of j^{th} neuron - Time-Steps set by the EA

T_{ONj} is the On-time of the j^{th} neuron - Time-Steps set by the EA

D_{2j} is the Off-Delay of the j^{th} neuron - Time-Steps set by the EA

T_{OFFj} is the Off-time of the j^{th} neuron - Time-Steps set by the EA

In all other respects, the behaviour of this model is the same as its biped counterpart. Using this upgraded neuron model, the quadruped can pause before moving its leg forward or backward. For example, assume such a neuron model is connected to the back left leg of a quadruped robot (refer to figure 7.5) and is responsible for the horizontal movements. The neuron has values assigned to its parameters and their expected output when simulated is shown in figure 7.12.

Neuron Parameters:
$D_1 = 0$
$T_{ON} = 10$
$D_2 = 80$
$T_{OFF} = 10$
$W = -1$

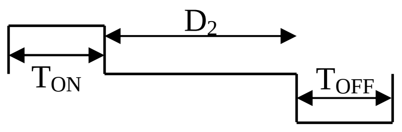


Figure 7.12 Expected Neuron output

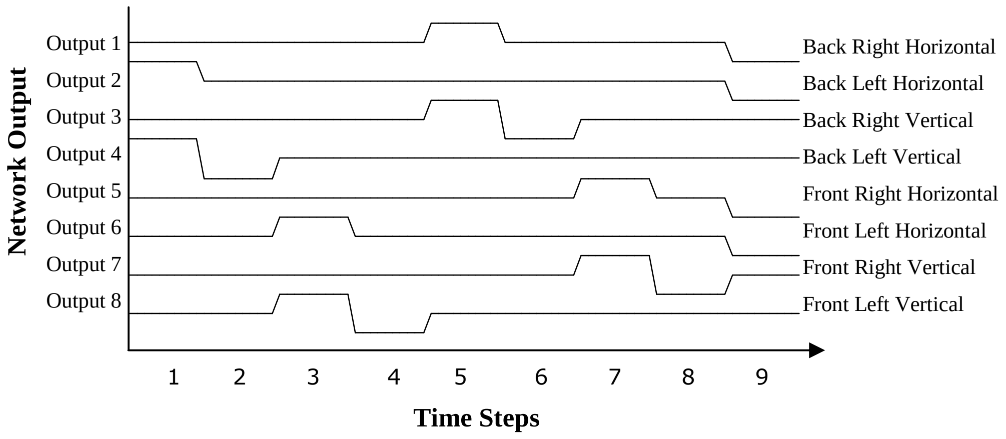
Assume initial Input = -1 Net(t) = Input x Weight = -1 x -1 Net(t) > 0 ⇒ **Output = 0** for $0 < t < D_1$ and as $D_1 = 0$ for this simulation **Output = 1** for $0 < t < T_{ON}$ and T_{ON} 10
As $T_{ON} = 10$, for the next 10 Time-Steps the **Output = +1** ⇒ Back left leg moves a unit horizontally forward, for next 10 Time-Steps as shown in figure 7.2.5 – frame (2)
When T_{ON} is up, **Output = 0** for $0 < t < D_2$ and $D_2= 80$
As $D_2 = 80$, for the next 80 Time-Steps the **Output = 0** ⇒ Back left leg hold its current position for next 80 Time-Steps as shown in figure 7.2.5 – from frame (3) to frame (9)
When D_2 is up, **Output = -1** for $0 < t < T_{OFF}$ and $T_{OFF} = 10$
As $T_{OFF} = 10$, for the next 10 Time-Steps the **Output = -1** ⇒ Back left leg moves a unit horizontally backward, for next 10 Time-Steps as shown in figure 7.2.5 – frame (10)

Thus, this upgraded neuron model could be used to control the quadrupeds.

Before considering the network topology, the expected output for a quadruped walk (as shown in section 7.2) is presented. This is done because it is easier for the reader to check that the output of the new network is correct. The neuron parameters are shown in figure 7.13 (a) and the network output in 7.13(b). The left side of the graph shows the neural outputs for each degree of freedom and the right side shows to which actuator the output of the neuron is connected. The output is shown for nine Time-Steps. Note that each output was calculated individually with time-values assigned.

	N1	N2	N3	N4	N5	N6	N7	N8
D_1	4	0	4	0	6	2	6	2
T_{ON}	1	1	1	1	1	1	1	1
D_2	3	7	0	0	1	5	0	0
T_{OFF}	1	1	1	1	1	1	1	1

(a) Neuron Parameters

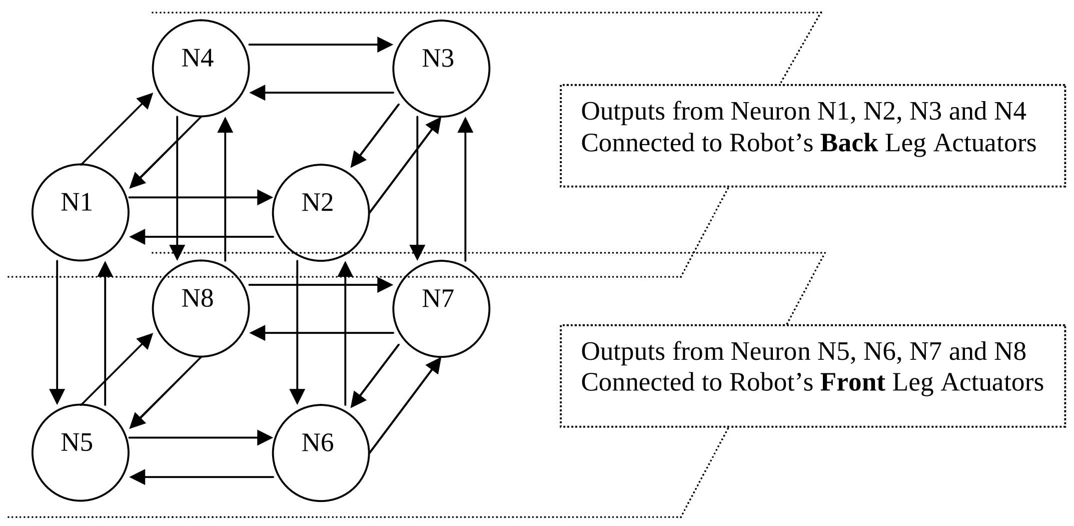


(b) Output

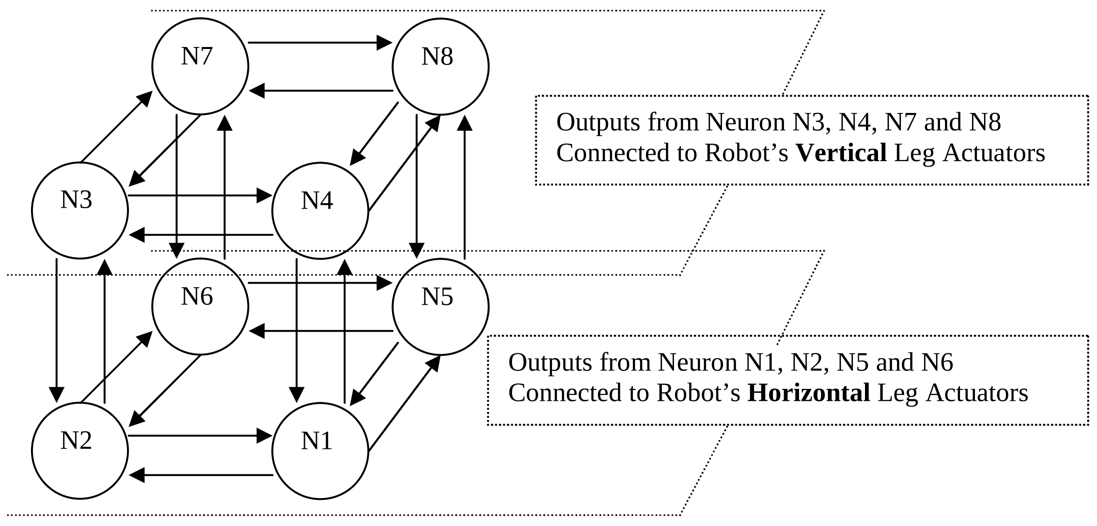
Figure 7.13 Expected Outputs for a Quadruped Robot

7.4.2 Network Topology

Initially (similarly to the bipeds) a simple recurrent network was designed as shown in figure 7.14 (a) and (b).


N1
N2
N3
N4
N5
N6
N7
N8
Outputs from Neuron N1, N2, N3 and N4 Connected to Robot's **Back** Leg Actuators
Outputs from Neuron N5, N6, N7 and N8 Connected to Robot's **Front** Leg Actuators

(a) Front/Back Network View


N1
N2
N3
N4
N5
N6
N7
N8
Outputs from Neuron N3, N4, N7 and N8 Connected to Robot's **Vertical** Leg Actuators
Outputs from Neuron N1, N2, N5 and N6 Connected to Robot's **Horizontal** Leg Actuators

(b) Vertical/Horizontal Network View

Figure 7.14 Initial Network Topology for Quadruped

Refer to figure 7.13(b) for each neuron's expected output and its corresponding degree of freedom (actuator) connection. Both diagrams in figure 7.14 show the same network, viewed in different direction, which may help to understand the network topology. The view in figure 7.14 (a) shows the network in two parts – upper-part neurons connected to the back two legs of the quadruped and the lower-part neurons connected to the front two legs. The view in figure 7.14 (b) shows the network from a different direction – upper layer neurons (N3, N4, N7 and N8) connected to the vertical degrees of freedom and lower layer neurons (N1, N2, N5 and N6) connected to the horizontal degrees of freedom. These views are shown to help visualise the network and its connections to the robot's actuators.

This network topology highlighted an issue regarding the number of connections of each of the neurons in the network. This arose because of what was learned from the biped network – that a network having more than one connection can cause instability. However, the extra connection in the biped network was the recurrent connection and not a connection from another neuron. Hence, to rule out the possibility of instability, a manual simulation was carried out for the network shown in figure 7.14. The network parameters are similar to those shown in figure 7.13 (a) with the addition that the weight for each connection was set to '-1'. The initial inputs to the neurons were also assigned to '-1'. As the numbers of neurons are larger than in the biped case, a step-by-step calculation of the network is difficult to show here. Hence, the initial inputs, weights, Nets, calculated outputs and new Nets are shown below in figure 7.15 and in figure 7.16. The Net for each neuron is calculated using equation 7.10.

Initial Inputs

i\j	1	2	3	4	5	6	7	8
1	0	-1	0	-1	-1	0	0	0
2	-1	0	-1	0	0	-1	0	0
3	0	-1	0	-1	0	0	-1	0
4	-1	0	-1	0	0	0	0	-1
5	-1	0	0	0	0	-1	0	-1
6	0	-1	0	0	-1	0	-1	0
7	0	0	-1	0	0	-1	0	-1
8	0	0	0	-1	-1	0	-1	0

Weights

i\j	1	2	3	4	5	6	7	8
1	0	-1	0	-1	-1	0	0	0
2	-1	0	-1	0	0	-1	0	0
3	0	-1	0	-1	0	0	-1	0
4	-1	0	-1	0	0	0	0	-1
5	-1	0	0	0	0	-1	0	-1
6	0	-1	0	0	-1	0	-1	0
7	0	0	-1	0	0	-1	0	-1
8	0	0	0	-1	-1	0	-1	0

N	1	2	3	4	5	6	7	8
Initial Nets	3	3	3	3	3	3	3	3

Figure 7.15 Initial Inputs, Weights and the Initial Nets of the Network

In figure 7.15, in the initial inputs' table, the left-most column is the number of each neuron and the top row numbers refers to the connecting neurons. A zero in the table shows that there is no connection between those neurons. Similarly, the weights are also shown above. This structured table helps in the calculation of the initial Nets. For example, the Net of the N1 is calculated by referring to the table.

Net-N1= $\sum(In_{ij} \text{ x } W_{ij})$ – The Input to the ith neuron from the jth connection can be seen from the table.

$$= (In_{12} \text{ x } W_{12}) + (In_{14} \text{ x } W_{14}) + (In_{15} \text{ x } W_{15})$$

$$= (-1 \text{ x } -1) + (-1 \text{ x } -1) + (-1 \text{ x } -1)$$

$$= 3$$

Thus, by calculating the Net values, the outputs are generated. The Nets are re-calculated for a neuron when its ON/OFF cycle is finished. The table shown in figure 7.16 shows such simulated network outputs in each Time-Step and the new Nets calculated.

Network Output										New Net							
	N1	N2	N3	N4	N5	N6	N7	N8		N1	N2	N3	N4	N5	N6	N7	N8
T1	0	1	0	1	0	0	0	0		x	x	x	x	x	x	x	x
T2	0	0	0	-1	0	0	0	0		x	x	x	x	x	x	x	x
T3	0	0	0	0	0	1	0	1		x	x	x	0	x	x	x	x
T4	0	0	0	0	0	0	0	-1		x	x	x	-1	x	x	x	x
T5	1	0	1	1	0	0	0	0		x	x	x	1	x	x	x	0
T6	0	0	-1	-1	0	0	0	0		x	x	x	x	x	x	x	-1
T7	0	0	1	1	1	0	1	1		x	x	1	1	x	x	x	1
T8	0	0	-1	-1	0	0	-1	-1		x	x	x	x	x	x	x	x
T9	-1	-1	1	1	-1	-1	1	1		x	x	2	2	x	x	2	2

Figure 7.16 Simulated Network Outputs and new Net

In the figure, the left-most column with T1, T2, etc, represents the Time-Steps and the top row with N1, N2, etc represents the neurons in the network. In the new Net table (at right of figure 7.16), the ‘x’ shows that the Net is not re-calculated. The calculated outputs were plotted on a graph and compared with the expected (wanted) output. This is shown in figure 7.17.

It is obvious from the above comparison that the simulated outputs did not match the expected outputs and this was due to instability caused by the connection patterns of the neurons. Some of the neurons triggered where they were not supposed to trigger.

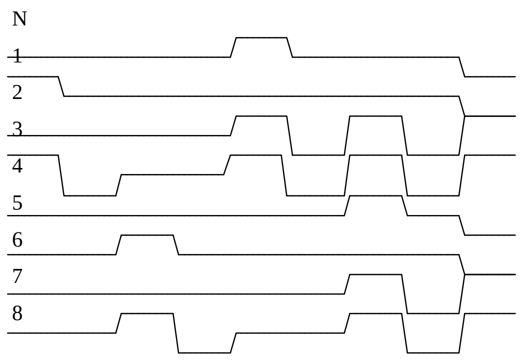
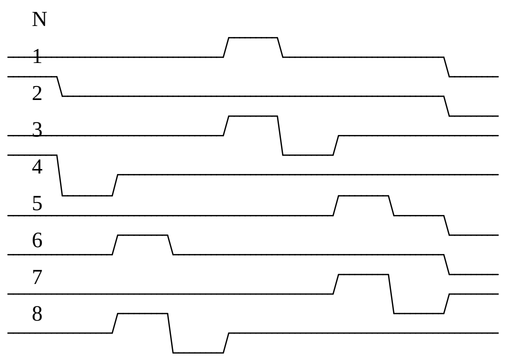
Simulated Network Output	Wanted Output
	

Figure 7.17 Comparison of Simulated and Wanted results

Of course an EA may be used to tune the weights but this may not be efficient because the bipeds already showed that the weights being independent from the time-parameters will cause non-linearity in the outputs. Thus, this neuron model and the topology described are difficult to use in real-time.

At this point of the project, further investigation into the literature showed that a neural network based on a Lamprey (investigated by Grillner et al.) might be a useful model [1].

7.4.3 Lamprey Network to Quadruped Network

A lamprey is a primitive fish. Its importance is that it is the only biological CPG/reflex system to be studied in detail. As such, it plays an important role in the biological underpinning of systems such as those being discussed here. It was logical therefore, to turn to it as inspiration when developing the form and operation of the quadruped system described here. Therefore, a short description of its operation is presented in the next few paragraphs before turning to the quadruped network and its biological inspiration. More about the operation of the network can be learned from reference [1]. The network topology is shown in figure 7.18.

The lamprey has its motor-neurons in segments on the right and left sides of the body. It swims by triggering alternate segments under the control of the brainstem. This method, of having the brainstem control and segmented motor-neuron, inspired a new model of network topology in the project. It also suited the neuron model and the problem faced in developing a network for the quadruped (refer to section 7.4.2). This approach led to the following two points:

1. The neurons are triggered by a central control system (the brainstem in the biological system).
2. Because of this simplified approach, the weights are unnecessary and the RTEA need only evolve the time-dependent parameters of the neurons.

In this project, the central control neuron was called the “Mother-Neuron” (MN) and the time-dependent neuron model which is controlled by the MN is called the “Child-Neuron” (CN).

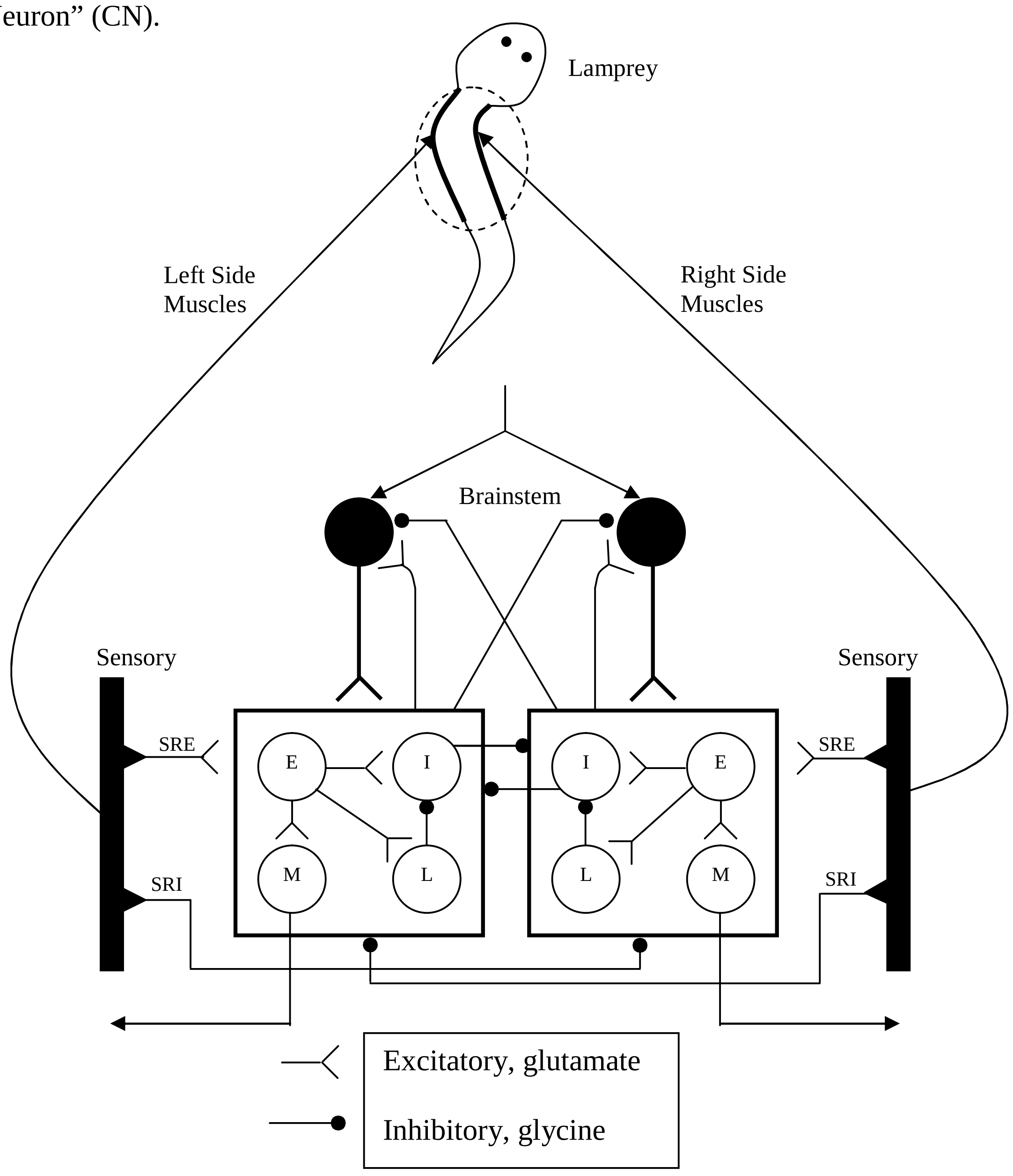


Figure 7.18 Lamprey, Segmental CPG network
(After Grillner et al.)

Figure 7.19 shows a schematic diagram of MN/CN and the resemblance of the lamprey's brainstem/motor-neuron model may be noticed.

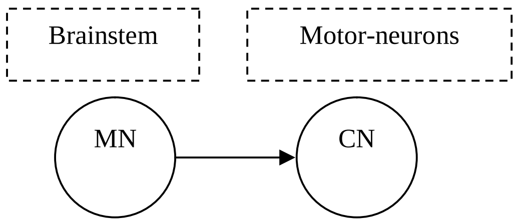
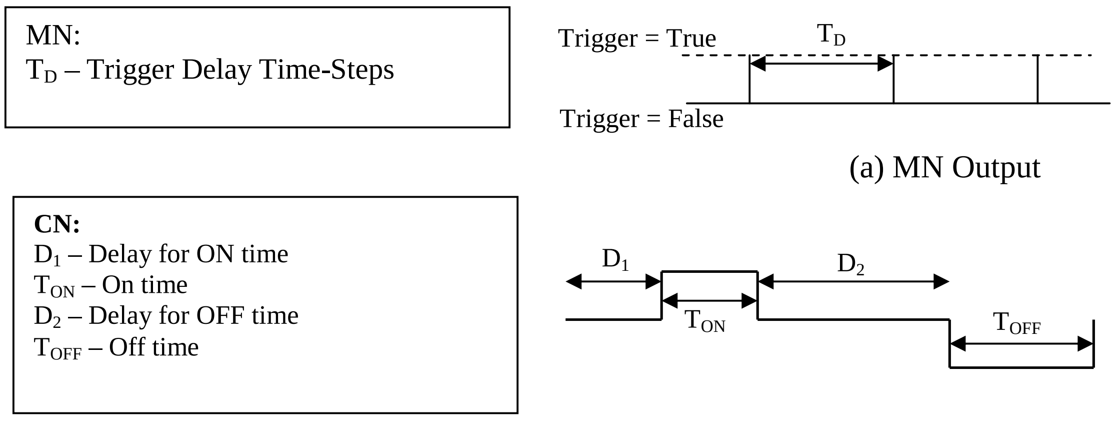


Figure 7.19 Resemblance of MN/CN and Lamprey's Brainstem/Motor-neuron

With this new proposal, the expected output when simulated is shown in figure 7.20.



(a) MN Output

(b) CN Output

Figure 7.20 Expected Output

The MN triggers the CN at T_D (for example, if T_D is four, then once in every four Time-Steps the MN sends a triggering signal to the CN). The activation function of the MN/CN is shown below in equation 7.11. The operation of the CN is straightforward; if it is triggered, it completes the ON/OFF cycle as described and, if it is not triggered, the output is set to zero until it receives a triggering signal from the MN.

$$\begin{aligned}
&if \quad (t = T_D) \Rightarrow MN(Tigger) = True \quad and \quad CN_j(Trigger) = True \quad and \\
&\qquad for \quad 0 < t < D_{1j} \quad : Out_j(t) = 0 \quad and \\
&\qquad for \quad 0 < t < T_{ONj} \quad : Out_j(t) = +1 \quad and \\
&\qquad for \quad 0 < t < D_{2j} \quad : Out_j(t) = 0 \quad and \\
&\qquad for \quad 0 < t < T_{OFFj} \quad : Out_j(t) = -1 \\
&else \qquad \Rightarrow MN(Trigger) = false \quad and \quad CN_j(Trigger) = False \quad and \\
&\qquad for \quad 0 < t < \infty \quad : Out_j(t) = 0
\end{aligned} \tag{7.11}$$

Where *MN* is the Mother-Neuron

T_D is the Time-Delay - Time-Steps set by the EA

CN_j is the j^{th} Child-Neuron

Out_j is the output of j^{th} neuron

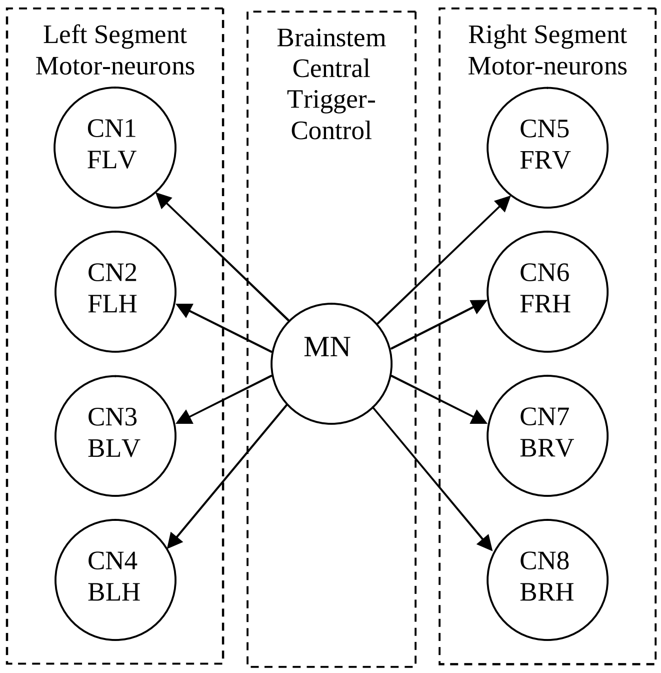
D_{1j} is the On-Delay of j^{th} neuron - Time-Steps set by the EA

T_{ONj} is the On-time of the j^{th} neuron - Time-Steps set by the EA

D_{2j} is the Off-Delay of the j^{th} neuron - Time-Steps set by the EA

T_{OFFj} is the Off-time of the j^{th} neuron - Time-Steps set by the EA

With the upgraded neuron model in hand, a detailed network topology was developed. The model lends itself to a simple feed-forward network, as shown in figure 7.21 and this was adopted.


Left Segment Motor-neurons
Brainstem Central Trigger-Control
Right Segment Motor-neurons
CN1 FLV
CN2 FLH
CN3 BLV
CN4 BLH
MN
CN5 FRV
CN6 FRH
CN7 BRV
CN8 BRH

CN1-FLV – Child-Neuron1 for Front-Left-Vertical actuator
CN2-FLH – Child-Neuron2 for Front-Left-Horizontal actuator
CN3-BLV – Child-Neuron3 for Back-Left-Vertical actuator
CN4-BLH – Child-Neuron4 for Back-Left-Horizontal actuator
CN5-FRV – Child-Neuron5 for Front-Right-Vertical actuator
CN6-FRH – Child-Neuron6 for Front-Right-Horizontal actuator
CN7-BRV – Child-Neuron7 for Back-Right-Vertical actuator
CN8-BRH – Child-Neuron8 for Back-Right-Horizontal actuator

Figure 7.21 MN/CN Initial Network Topology

The next task was to set the time-parameters of the MN/CN manually and simulate the network to test if it gives a suitable predictable output. The CN test parameters and the expected output were the same as shown in section 7.4.1 (refer to figure 7.13 (a) and (b)). The MN test parameter is shown below.

MN Parameter:
T_D = 9 Time-Steps

The MN was set to trigger all the neurons once in every nine Time-Steps. The network was simulated and the result was compared with the expected result. This is shown below in figure 7.22.

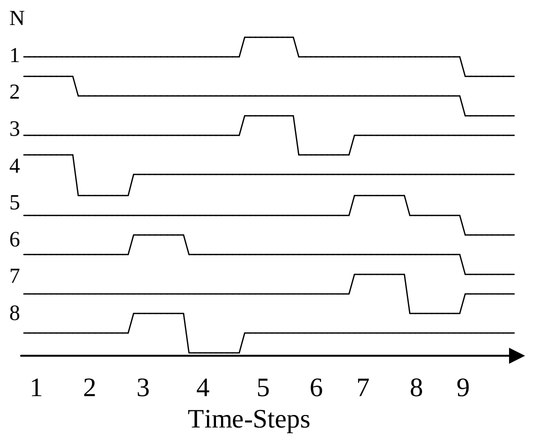
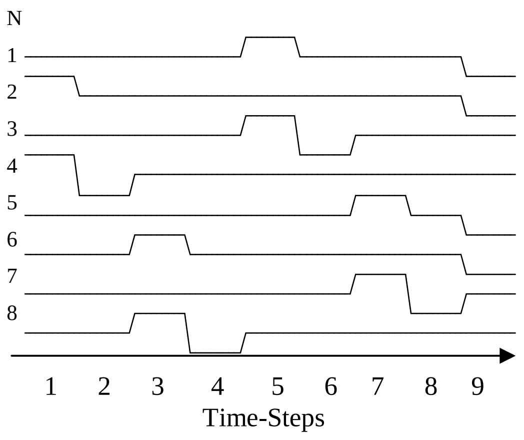
Simulated Result	Expected Result
	

Figure 7.22 Comparison of Simulated and Expected result

The comparison above shows that the network generated the expected output. However, there was a problem. As the MN is controlling all the CNs, if any of these neurons, which has finished its ON/OFF cycle, has to start the cycle again, it cannot. For example, in the simulation, CN4 has finished the ON/OFF cycle by Time-Step 2 (refer to figure 7.22). It cannot trigger again until the MN sends the next triggering signal. This, in real-time, is not an efficient way of controlling the CNs because the robot’s legs which are controlled by CN parameters, may be subject to changes at anytime according to the conditions underfoot. This means that one CN cannot wait to be triggered by the MN if the CN is required to generate an output. This flexibility issue was carefully investigated. The next section explains the enhancement of flexibility in the MN/CN network topology.

7.4.4 Enhancement of Flexibility – MN/CN and Network Topology

Using the lamprey network as inspiration, the segmented network was developed further. The lamprey's left segment and the right segments perform similar actions and are controlled by the central brain stem. This idea was used to develop a central trigger-control (MN) for similar actions (vertical/horizontal neurons - CN) of the front and back legs of the quadruped. Using this idea the following network topology was developed and is shown in figure 7.23.

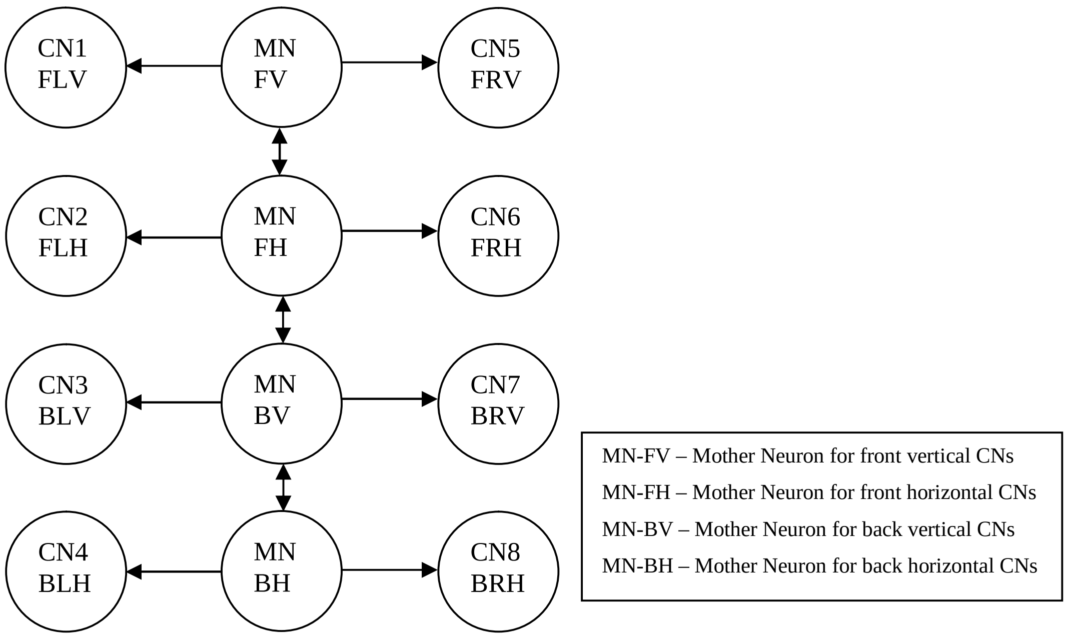

CN1 FLV
MN FV
CN5 FRV
CN2 FLH
MN FH
CN6 FRH
CN3 BLV
MN BV
CN7 BRV
CN4 BLH
MN BH
CN8 BRH
MN-FV – Mother Neuron for front vertical CNs
MN-FH – Mother Neuron for front horizontal CNs
MN-BV – Mother Neuron for back vertical CNs
MN-BH – Mother Neuron for back horizontal CNs

Figure 7.23 Upgraded Quadruped Network

The weights between the MNs are set to one (defining that the connections are turned on) as a default. When any of the MNs is triggered from the higher brain functions (as discussed in chapter 2, section 2.5), it triggers the other mother neurons connected to it. Also, the algorithms operating on the network can turn off any one of the MN connections to make the corresponding part of the network passive (if the conditions underfoot require it). Note that this weight may be the subject of future investigation; for this project only the time-parameters were evolvable.

As the CN's ON/OFF times (refer to the CN parameters in figure 7.13a) are different for different CNs, co-ordinating the triggering action with such a network (shown in figure 7.23) was the next task. The parameters were initially set manually to generate walk patterns for the quadruped robot.

In the task of co-ordinating the triggering time, initially the MN was set as a clock, which can cycle continuously. The CN was altered so that it no longer has to finish the ON/OFF cycle when triggered by the MN; when the CN receives the trigger-ON signal, it only has to finish its ON cycle and wait for the next trigger-OFF signal to start its OFF cycle. The splitting of ON and OFF cycles helps the co-ordination of the MN and the two CNs connected to it. As the ON and OFF cycles were now triggered individually, with different timings, new time-parameters were introduced to the MN. This is shown below with its output shown in figure 7.24.

MN Time-Parameters:

T_{ON} – On-time

T_{OFF} – Off-time

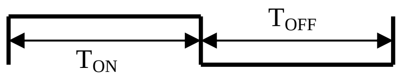


Figure 7.24 Output of MN

The CNs new activation function is given in equation 7.11.

$$\begin{aligned}
&if \quad (MN \xrightarrow{Sends\ ON\ Signal} CN_j) \Rightarrow for \quad 0 < t < D_{1j} \quad : Out_j(t) = 0 \quad and \\
&\qquad\qquad\qquad for \quad 0 < t < T_{ONj} \quad : Out_j(t) = +1 \quad and \\
&\qquad\qquad\qquad for \quad T_{ONj} < t < \infty \quad : Out_j(t) = 0 \\
&if \quad (MN \xrightarrow{Sends\ OFF\ Signal} CN_j) \Rightarrow for \quad 0 < t < D_{2j} \quad : Out_j(t) = 0 \quad and \\
&\qquad\qquad\qquad for \quad 0 < t < T_{OFFj} \quad : Out_j(t) = -1 \quad and \\
&\qquad\qquad\qquad for \quad T_{OFFj} < t < \infty \quad : Out_j(t) = 0
\end{aligned} \tag{7.11}$$

Where *MN* is the Mother-Neuron

CN_j is the j^{th} Child-Neuron

Out_j is the output of j^{th} neuron

D_{1j} is the On-Delay of j^{th} neuron - Time-Steps set by the EA

T_{ONj} is the On-time of the j^{th} neuron - Time-Steps set by the EA

D_{2j} is the Off-Delay of the j^{th} neuron - Time-Steps set by the EA

T_{OFFj} is the Off-time of the j^{th} neuron - Time-Steps set by the EA

When the CN receives an ON signal from the MN, the CN generates an output of zero for D_1 Time-Steps and then an output of one for T_{ON} Time-Steps. When D_1/T_{ON} Time-Steps are up, it generates an output of zero until it receives the next OFF signal from the MN. Similarly, the OFF cycle of the CN is completed. The CN's ON/OFF cycle can be activated at any point of time by adjusting the MN's ON/OFF time and thus it is more flexible. As mentioned earlier, for the network shown in figure 7.23, the parameter values were manually set to generate walking patterns. A network with walk parameters is shown in figure 7.25.

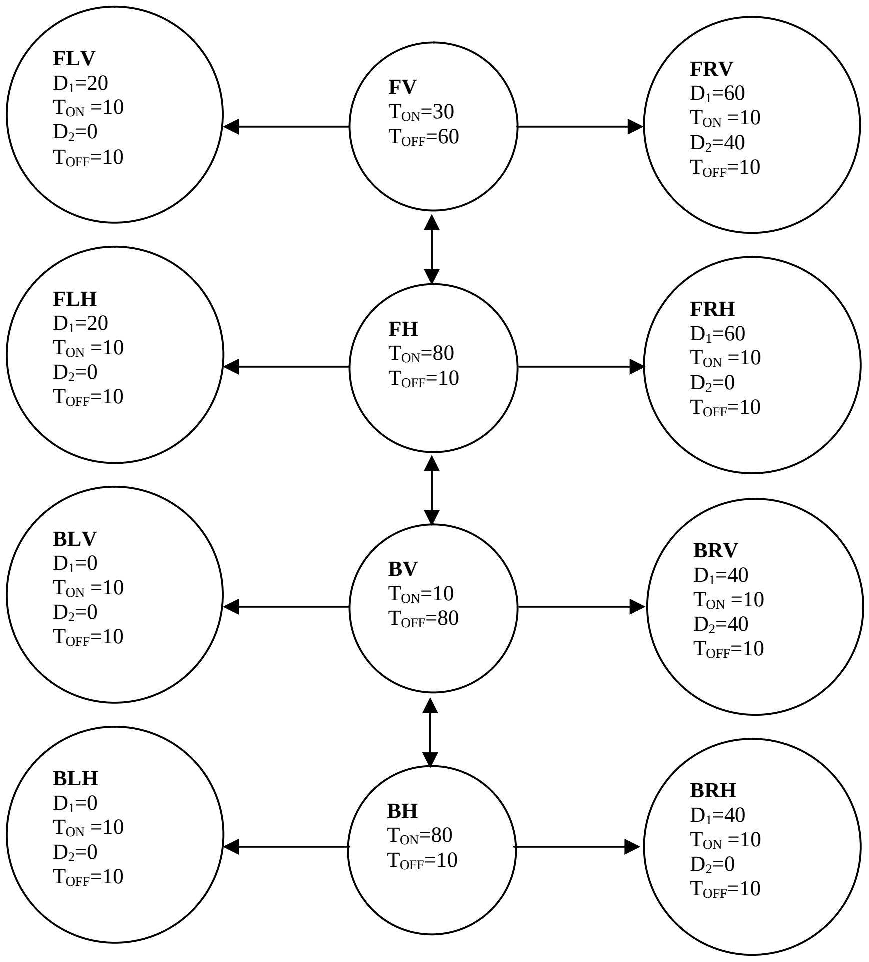
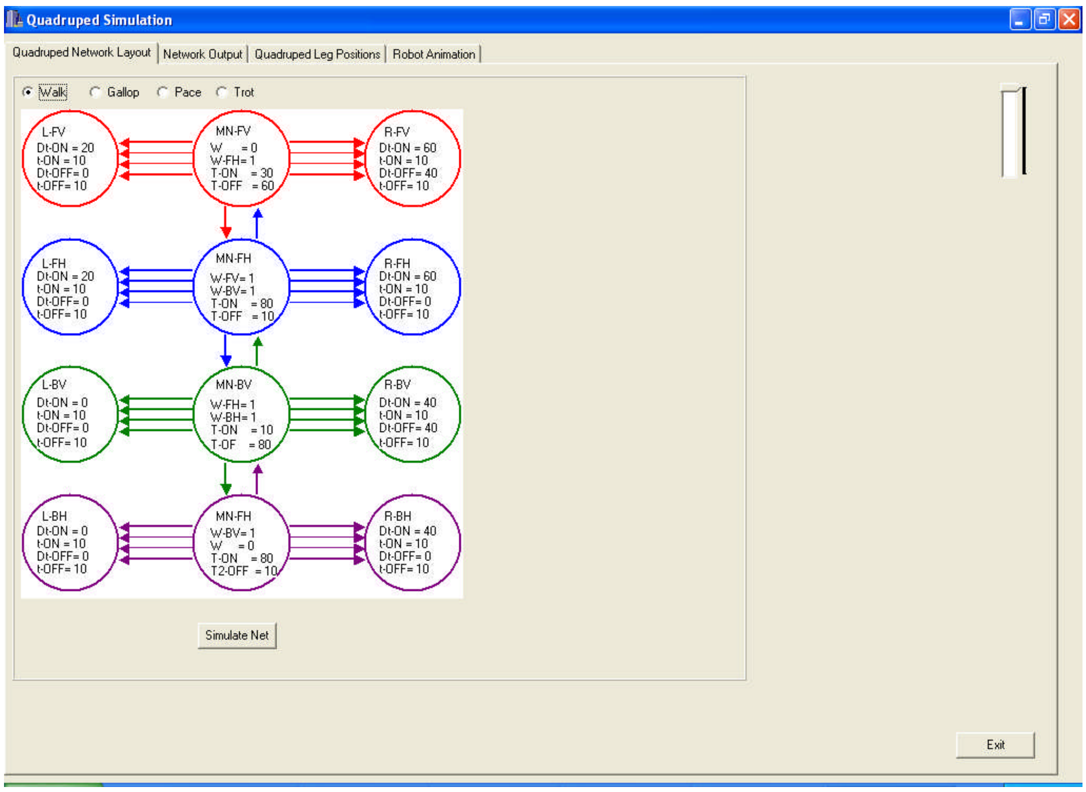


Figure 7.25 MN/CN Network – Walk Parameters

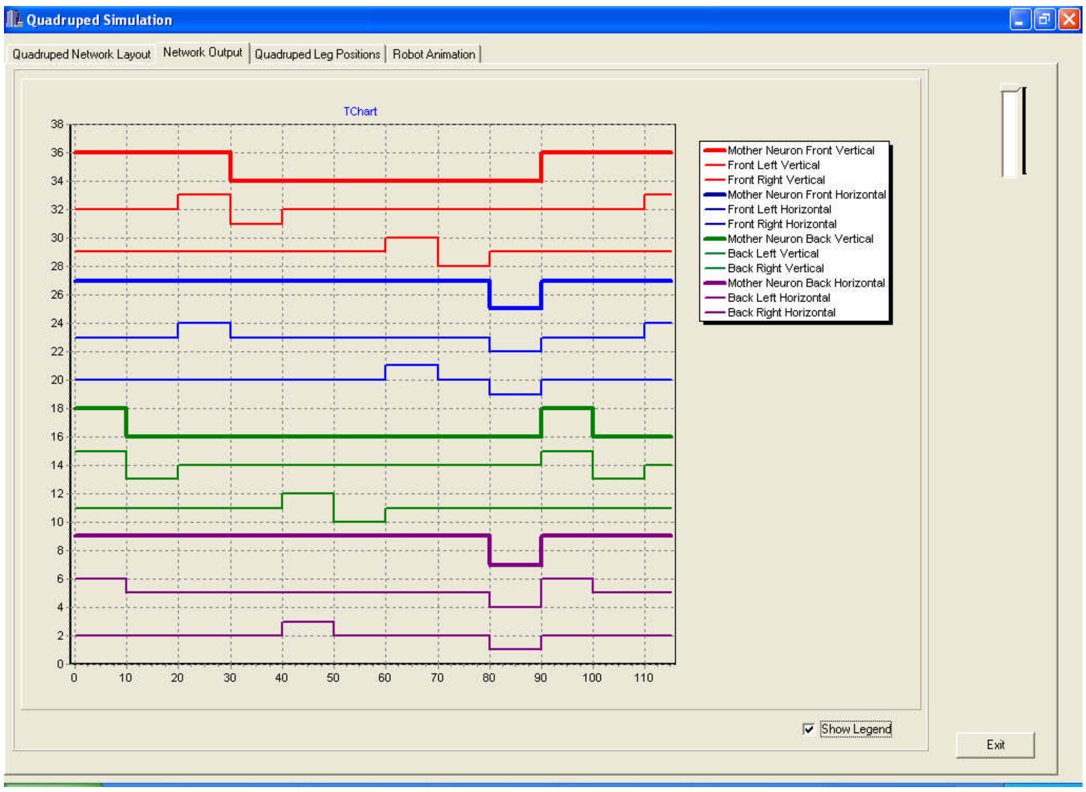
The network above was simulated and its output used to control the quadruped robot. The snap-shots of the simulation are shown in figure 7.26.

Apart from the walk parameters, other gaits like gallop, trot and pronk were set manually and tested. The network outputs for those gaits along with the snap-shots of the simulation are attached in appendix-D. This showed that the network is flexible enough to operate in real-time and so a system was finally ready to test with the RTEAs, discussed in chapter 4.


Quadruped Simulation
Quadruped Network Layout
Network Output
Quadruped Leg Positions
Robot Animation
L-FV Dt-ON = 20 t-ON = 10 Dt-OFF= 0 t-OFF= 10
MN-FV W = 0 W-FH= 1 T-ON = 30 T-OFF = 60
R-FV Dt-ON = 60 t-ON = 10 Dt-OFF= 40 t-OFF= 10
L-FH Dt-ON = 20 t-ON = 10 Dt-OFF= 0 t-OFF= 10
MN-FH W-FV= 1 W-BV= 1 T-ON = 80 T-OFF = 10
R-FH Dt-ON = 60 t-ON = 10 Dt-OFF= 0 t-OFF= 10
L-BV Dt-ON = 0 t-ON = 10 Dt-OFF= 0 t-OFF= 10
MN-BV W-FH= 1 W-BH= 1 T-ON = 10 T-OF = 80
R-BV Dt-ON = 40 t-ON = 10 Dt-OFF= 40 t-OFF= 10
L-BH Dt-ON = 0 t-ON = 10 Dt-OFF= 0 t-OFF= 10
MN-FH W-BV= 1 W = 0 T-ON = 80 T2-OFF = 10
R-BH Dt-ON = 40 t-ON = 10 Dt-OFF= 0 t-OFF= 10
Simulate Net
Exit

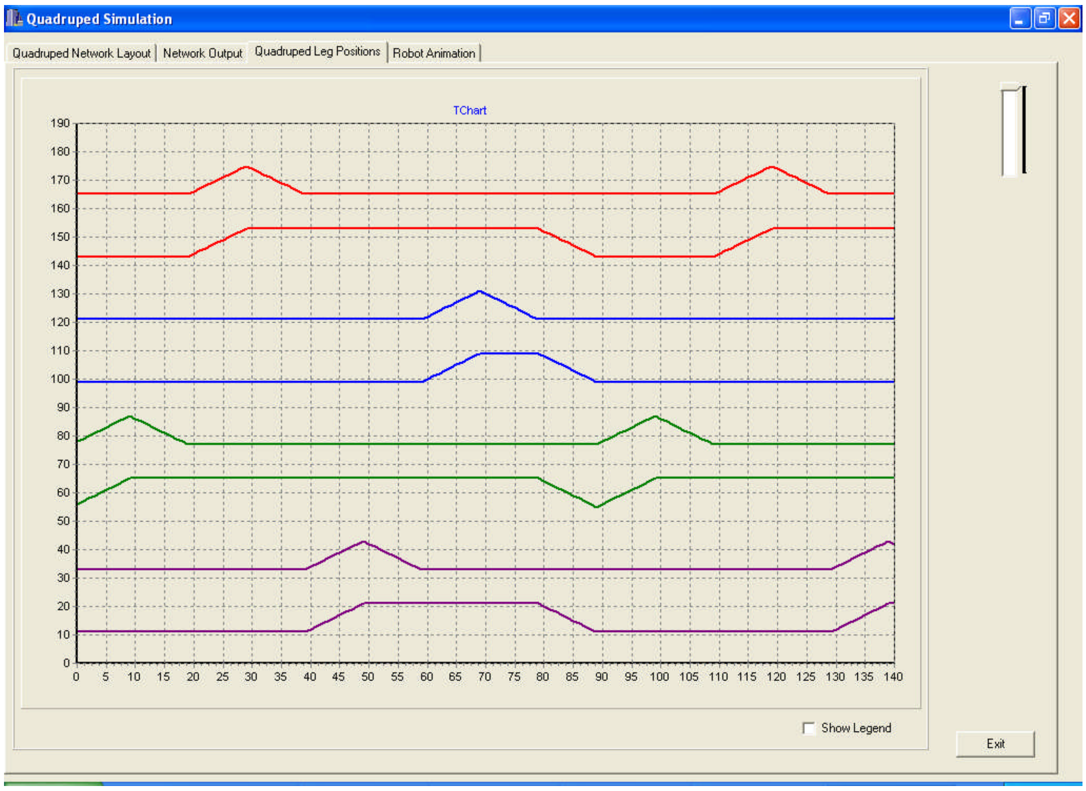
[✓ Walk] Gallop Pace Trot

Figures 7.26 (a) Network Parameters

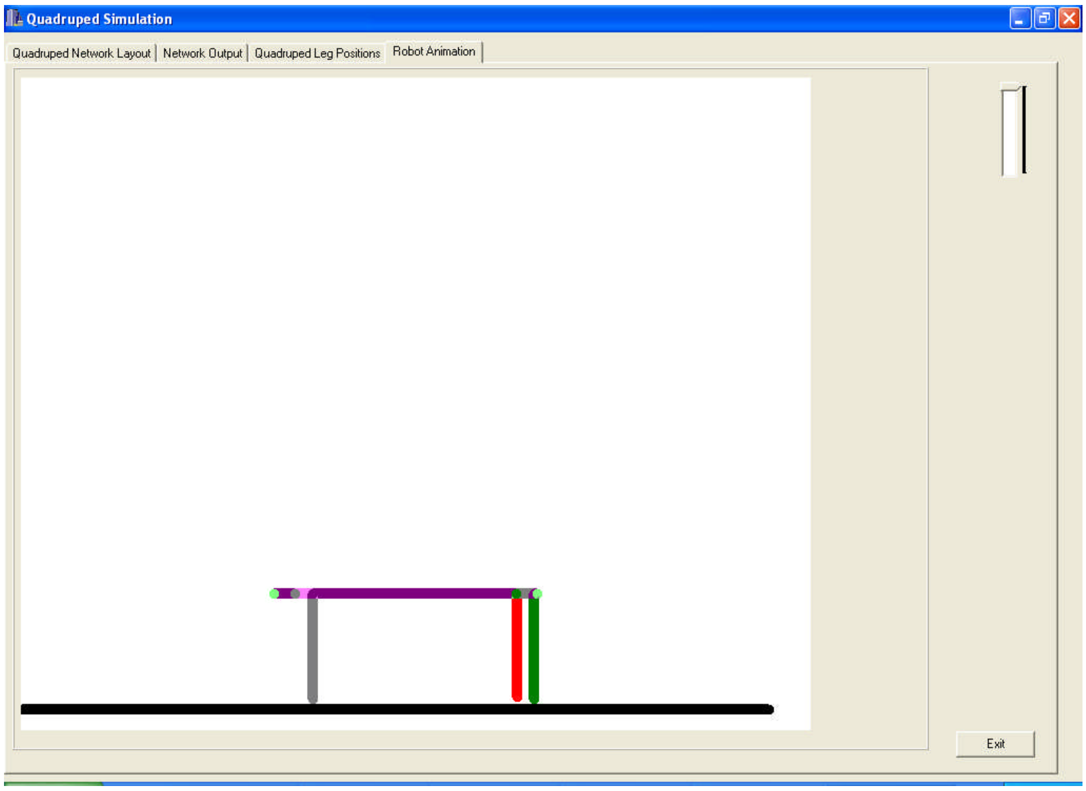


[✓] Show Legend

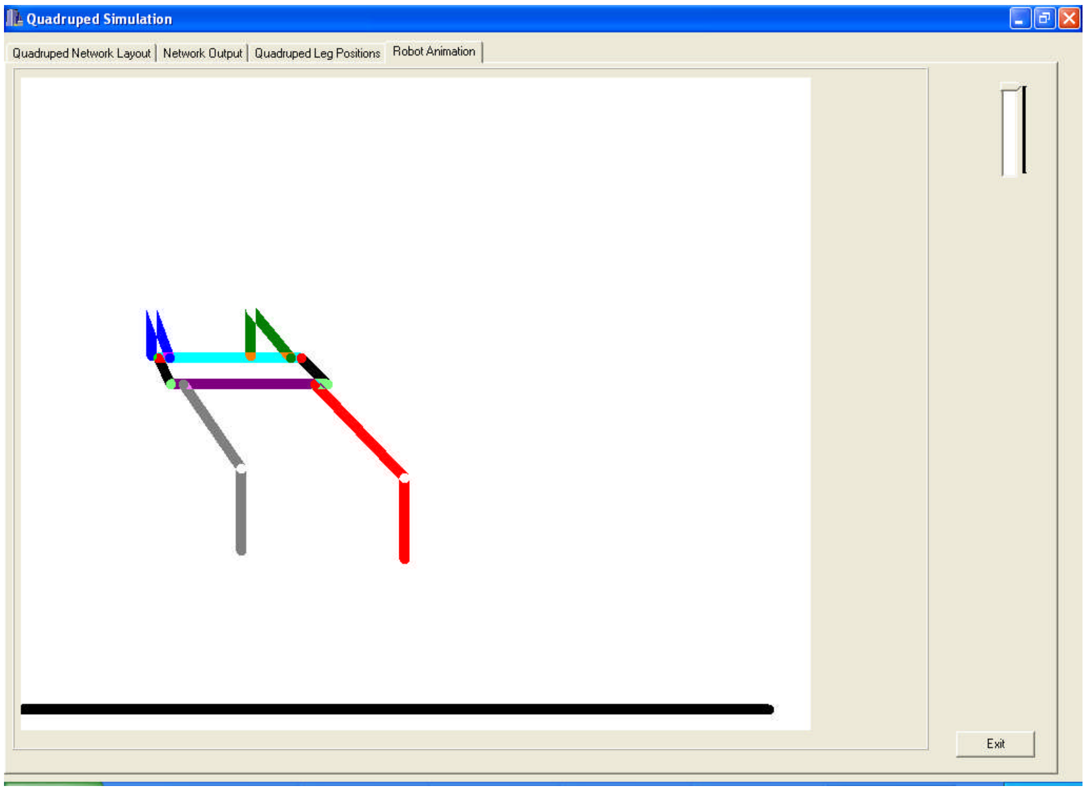
Figures 7.26 (b) Network Output



Figures 7.26 (c) Quadruped Leg Positions



Figures 7.26 (d) Side View of Quadruped Robot



Figures 7.26 (e) 3D View of Quadruped Robot

7.5 Summary

This chapter started by describing the design of the quadruped simulator and showing the complexity involved in controlling the quadruped. Then the fitness function was explained, which covered the three different parameters – stability, distance and fuel-efficiency. The neural network development was discussed and it was demonstrated how the lamprey network topology was used as an inspiration for the development of the MN/CN type network. This also covered the time-dependent neuron model development and showed how it can be flexible in real-time.

Chapter 8

RTEAs Operators

8.1 Introduction

In this chapter, the effects of different mutation operators under different experimental conditions are studied. The chapter starts by explaining the experimental parameters. The general flow (execution method) of the experiments is discussed. The results of these experiments are then highlighted, supported by graphs and charts. Finally a summary of the results is given.

8.2 Experimental Parameters

Testing started with the quadruped robot discussed in previous chapters. It was assumed to perform its actions on a flat surface. The RTEAs used were set up with different experimental parameters. These are shown in figure 8.1.

Random Number Type	**Condition of Acceptance**	**% of Equilibrium Disturbance**	**% of Mutation Size**	**% of System Disturbance**	**% of Required Stability**	**% of Required Distance**	**% of Required Fuel**
1/5 Rule	Simple	25%	25%	25%	25%	25%	25%
Normal		50%	50%	50%	50%	50%	50%
Uniform	SA	75%	75%	75%	75%	75%	75%
		100%	100%	100%	100%	100%	100%

Figure 8.1 Experimental Parameters

For each experiment, a parameter from each column of the above table (figure 8.1) was taken and simulated. The "1/5 Rule" is highlighted in the table because it uses a slightly different experiment setup from the others. This is explained in the discussion of "1/5 Rule" results.

In the table above, the "Random Number type", "Condition of Acceptance" and "% of Mutation Size" were already explained in chapter 4 (refer to section 4.2.3, 4.2.4 and 4.2.5). The other columns are explained below.

8.2.1 Equilibrium Disturbance

Before explaining about this experimental parameter, a brief explanation about how the network parameters are set up, before starting the experiment, is given. The robot’s fitness was configured to allow it to move forward by penalising any other movement. So, instead of evolving the MN/CN parameters from scratch, the network was preset with walk-parameters. The MN parameters were ***not*** disturbed (that is, moved away from this preset positions); only the CN parameters were disturbed (randomly altered) by an amount specified by the experimental parameter – “% of Equilibrium Disturbance”. The idea was that this parameter would help the investigation of how far the system can be taken out of equilibrium (its known high-fitness state) and still evolve back efficiently. The following example is given to explain how this parameter is used.

The maximum range which a parameter can be disturbed was set to 10. Assuming that “% of Equilibrium Disturbance” = 25%. Then the disturbing range would be (10 x 0.25) = 2.5 ≈ 2 (fraction values are ignored). If the CN1’s original T_{ON} was 4, then a uniformly distributed random-number between minus and plus two (-2 and +2) is generated and added to T_{ON}. Thus, CN1’s equilibrium is randomly disturbed according to the value of the experimental parameter “% of Equilibrium Disturbance”.

8.2.2 System Disturbance

This parameter is used to set how many of the neurons are to be disturbed, as described above. For example, if it is 25%, then out of 8 neurons in CN, 2 are disturbed. This helps to understand to what extent (how many neurons) the RTEAs can efficiently optimise after disturbance. In order to analyse this effectively, it was made sure in each experiment that the number of vertical-neurons and horizontal-neurons selected for disturbing are equal. Also, this helps in studying the effects of the fitness parameters; Stability, Distance and Fuel which depend on both vertical and horizontal neurons. The way in which the vertical and horizontal neurons affect the fitness parameters is described in section 7.3.

8.2.3 Required Stability, Required Distance and Required Fuel

These parameters are used in the fitness function (refer to chapter 7, section 7.3, equation 7.4). They are assumed to simulate the physical environment in which the robot is placed. The example below is used to explain one such environment.

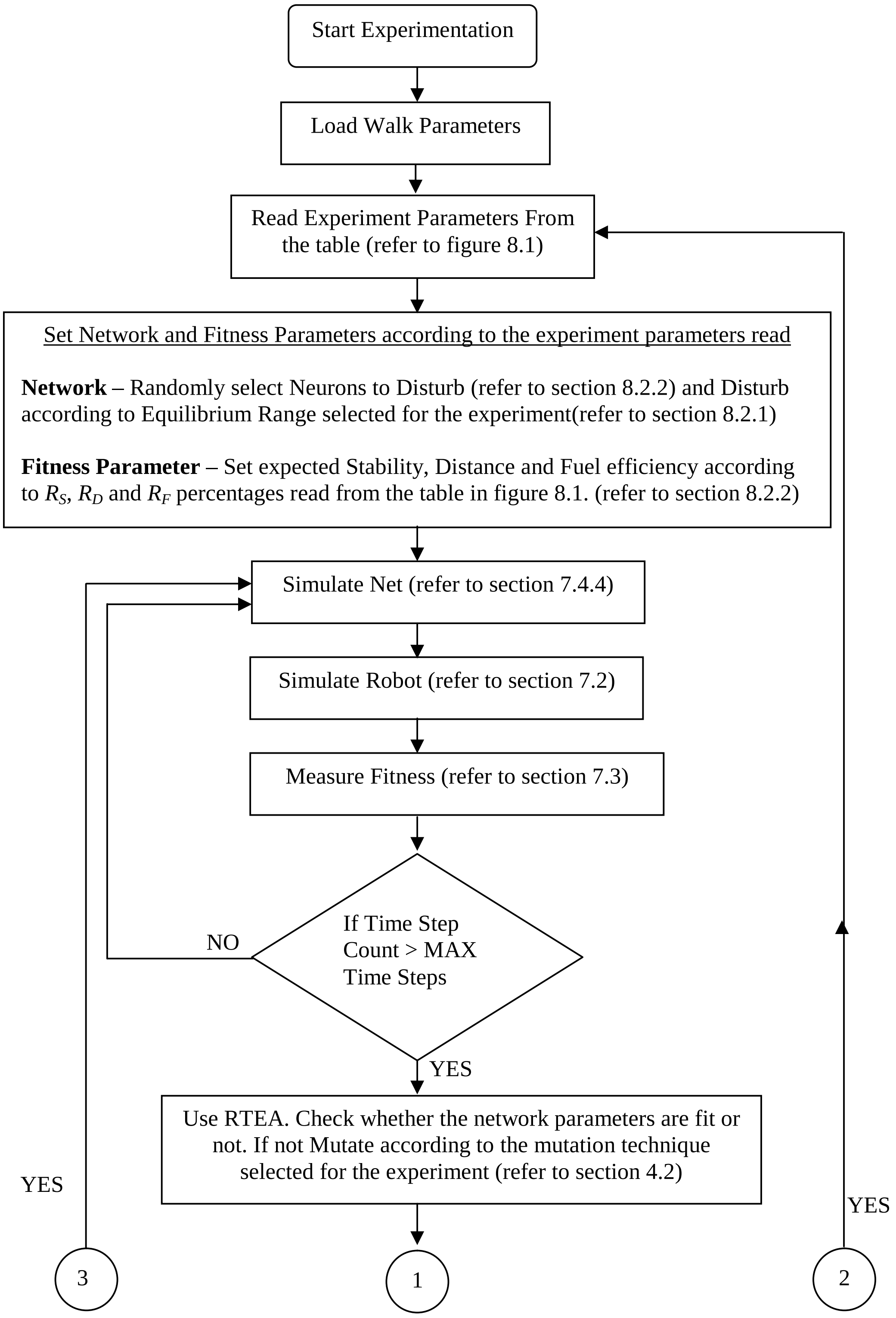
$$R_S = 100\%$$
$$R_D = 25\%$$
$$R_F = 50\%$$

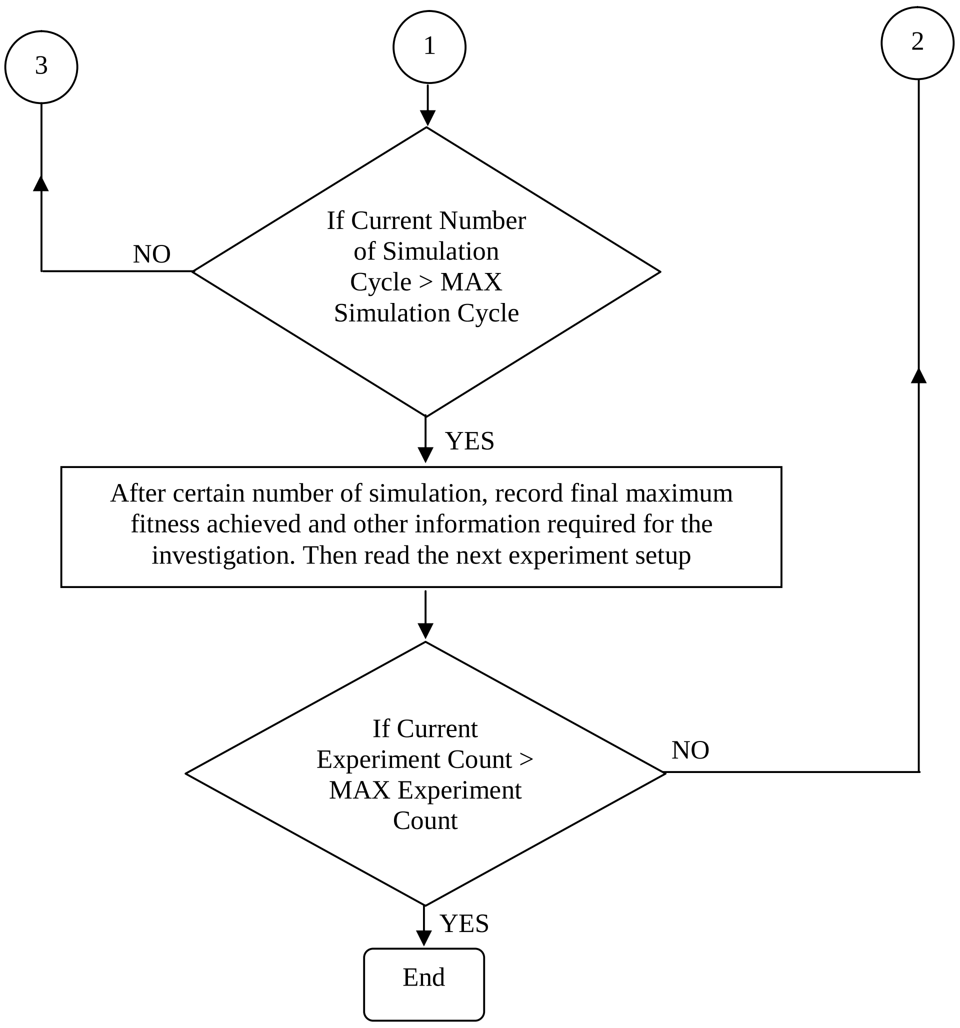
With such a setup, the robot could possibly be climbing a stoney-steep-hill. When climbing such hill, the robot normally has to be careful with stability (R_S – 100%), would not worry too much about the distance covered (R_D – 25%) and would not be much concerned about fuel used (R_F – 50%). Climbing without falling would be the top priority. Thus, by varying these fitness parameters, virtual environments for the robot may be created and this helps the investigation to analyse if the RTEAs were efficient in evolving the network parameters under different constraints. For every experiment the expected fitness of stability, distance and fuel were pre-calculated according to R_S, R_D and R_F selected; then the disturbed network is allowed to evolve by the RTEA to the expected fitness.

8.3 Testing Evolutionary Operators

Different mutation techniques – MA, MO and MS were discussed in chapter 4 (refer to section 4.2.2). Each mutation technique was tested against the experimental parameters discussed above. The total number of different combinations of experimental setup for the MA and MO techniques was ***18462***. Experimenting with the MS-technique did not use the experimental parameter "% of System Disturbance", because it was considered when the MS-technique is applied symmetrically (as discussed in chapter 4, section 4.2.2-Mutate Some) to the network; therefore, all the parameters have to be disturbed so that the effect of MS can be effectively studied. Hence the total number of experiments for the MS-technique was ***4608***.

As it can be seen, with the huge number of experiments, it is impossible to show the results of each of them. Anyway, the aim of this investigation is to learn which techniques with what experimental setup can be effective in real-time. This can be studied by taking the maximum fitness achieved in each experiment and taking an average of similar experiments. This is explained further along with the discussion of results. The general flow of experiments is shown in figure 8.2.


Start Experimentation
Load Walk Parameters
Read Experiment Parameters From the table (refer to figure 8.1)
Set Network and Fitness Parameters according to the experiment parameters read
Network – Randomly select Neurons to Disturb (refer to section 8.2.2) and Disturb according to Equilibrium Range selected for the experiment(refer to section 8.2.1)
Fitness Parameter – Set expected Stability, Distance and Fuel efficiency according to R_S, R_D and R_F percentages read from the table in figure 8.1. (refer to section 8.2.2)
Simulate Net (refer to section 7.4.4)
Simulate Robot (refer to section 7.2)
Measure Fitness (refer to section 7.3)
If Time Step Count > MAX Time Steps
NO
YES
Use RTEA. Check whether the network parameters are fit or not. If not Mutate according to the mutation technique selected for the experiment (refer to section 4.2)
YES
YES
3
1
2



3
1
2
If Current Number of Simulation Cycle > MAX Simulation Cycle
NO
YES
After certain number of simulation, record final maximum fitness achieved and other information required for the investigation. Then read the next experiment setup
If Current Experiment Count > MAX Experiment Count
NO
YES
End

Figure 8.2 General Flow Chart of Experimentation of RTEAs

*****The results discussed in this chapter are the result of simulating each experiment for 1000 cycles

As said earlier, the method shown above was coded using Borland C++ Builder and further description of the software is given in appendix-C.

8.3.1 Normal Distribution versus Uniform Distribution

Firstly, performance of Normal and Uniform random number distribution was compared by taking the overall average of all the experiments conducted for each of these operators. Typically, the total number of experiments for each operator with "Simple" or "SA" Conditional Acceptance was 4096. For example, for "Normal Distribution" with "Simple" acceptance, the total number of experiments is

(Equilibrium Range) x (Mutation Size) x (System Disturbance) x (Stability) x (Distance) x (Fuel) which gives a total of 4096 (refer table in figure 8.1). Hence, the results shown in each bar-chart in this section is an average of 4096 experiments.

Every bar-chart shown in figure 8.3, 8.4 and 8.5 has a common chart-legend showing two categories, "Average Expected" and "Average Evolved". This legend is shown in first figure 8.3 and it is not repeated as it is the same for all the discussions of MO and MS. The bars with the mesh grid type of fill are the total average of maximum fitness expected out of the experiments conducted. The dark bars with white dots are the total average of evolved fitness (evolved by the EA).

<u>Mutate All Result:</u>

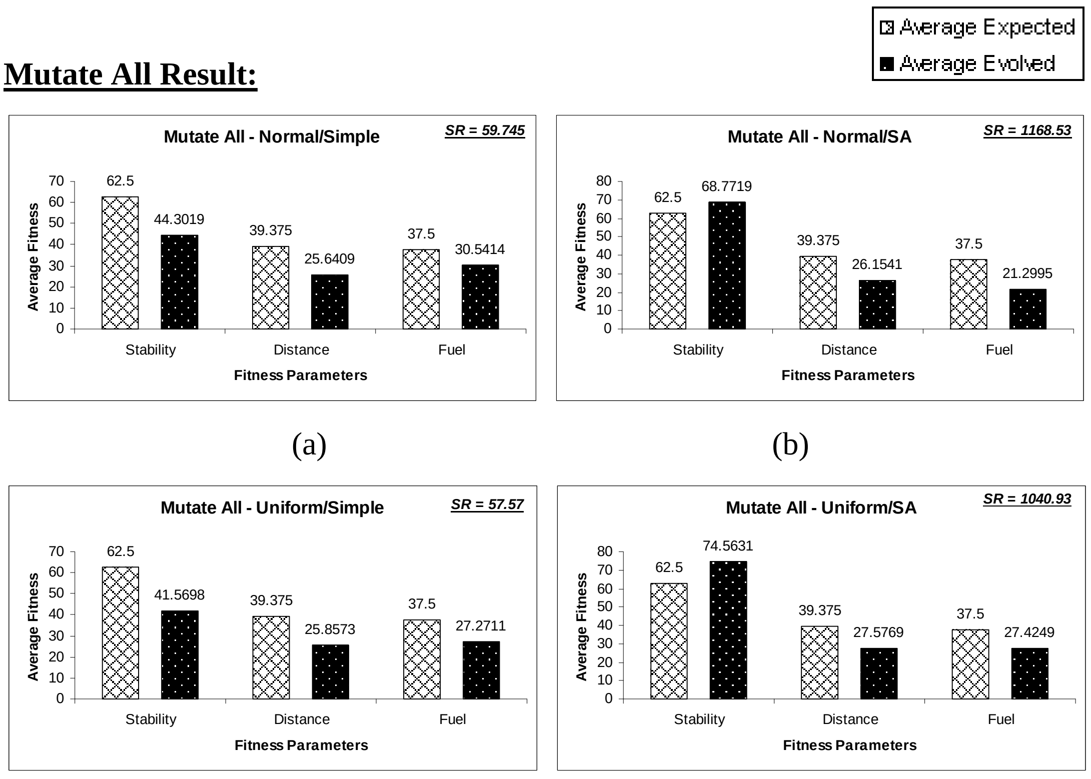


Figure 8.3 Mutate All – Normal Versus Uniform Distributions

The results of Normal Distribution with "Simple" acceptance are shown in figure 8.3(a) and with "SA" acceptance in figure 8.3(b). Similarly, the results of Uniform Distributions are shown in figure 8.3(c) and 8.3(d). In each chart, the Success-Rate (SR) is shown in the top-right corner. The SR is a measure of the rate of the number of times the system successfully evolves or moves closer to the required fitness. It is measured using the equation shown in 8.1.

$$SR = \frac{Number \quad of \quad Success}{Number \quad of \quad Simulation \quad Cycles} \tag{8.1}$$

The average fitness of each of stability, distance and fuel in each case for Normal and Uniform Distributions under "Simple" acceptance was close to each other, as can be seen in figure 8.3(a) and 8.3(c). Similarly, under "SA", the average fitness was similarly close to the others as well, which can be seen in figure 8.3(b) and 8.3(d). Also, the success rates of normal and uniform distributions were similar.

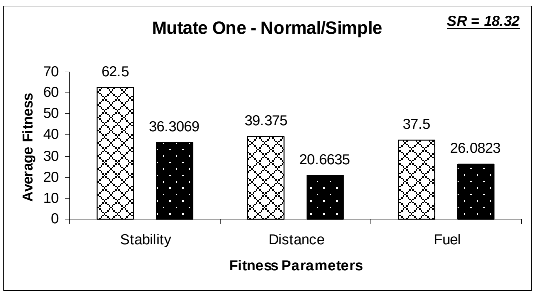
The probable reason that the Normal and Uniform Distribution performance was similar, was that the system, in which the operators are tested, requires a change in parameters with at least a difference of one or minus one (as a whole number). For example, consider the neuron activation equation 7.11. In the equation T_{ONj} decides how long the neuron has to stay ON and produce positive pulse. If T_{ONj} is two it produces two units of positive pulses. If it has to give out three units of positive pulses, then T_{ONj} has to change its value from two to three and no fractional change would help to give out three positive pulses.

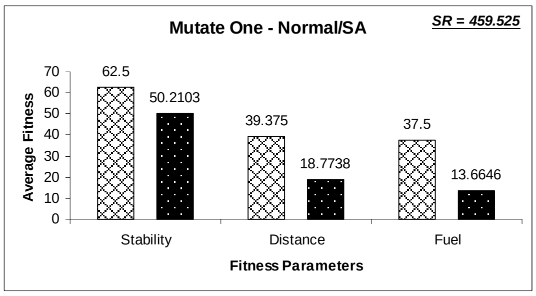
The Normal random number in this project generates fractional numbers depending on the Mutation Size (refer to section 4.2.3). The random number generated during the mutation process could be 0.512 which could have changed the T_{ONj} from 2 to 2.512. Clearly this will not satisfy the need for three positive pulses and the SR would remain the same. In the next cycle of mutation, it may generate a random value 0.488 which makes the T_{ONj} three. Then the SR is incremented.

The Uniform random number generator function, generates whole numbers depending on the mutation size (refer to section 4.2.3). The random number generated during the mutation process could have been one which changes T_{ONj} from 2 to 3 and the SR is incremented. Thus the two mutation operators may have ended up producing similar results of fitness and SR.

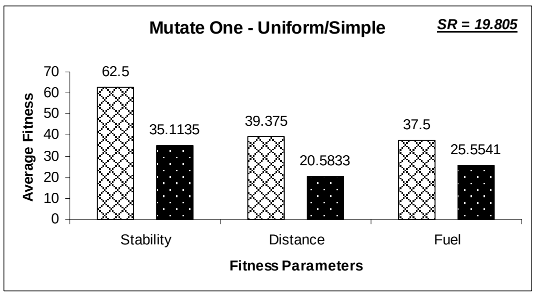
However, both the operators under SA have shown that the SR and the average fitness are higher than Simple acceptance. This is because the SA allows the error to increase. The more room there is for error, the more the room for finding improvement in fitness and hence higher SR. For example, consider two neurons which are undergoing a mutation process. In one cycle, both the neurons may have succeeded at getting close to the expected fitness. In the next cycle one of the neurons may have failed to improve its fitness and this may result in accepting the error. This, under SA, would change the other fit neuron and it would have to undergo the mutation process again, looking for an improvement (and finally it may have succeeded). Thus, more error causes more room for improvement and hence higher SR. This is a slight disadvantage when using the MA technique as it may disturb the fit parameters under SA acceptance. Having said that, only by using SA, the expected fitness (or at least close to expected) was reached which can be seen in the figure 8.3(b) and 8.3(d). This disadvantage can be avoided when the MO technique is used. This is discussed in the next part of this section. Overall, in this discussion it can be said that the performance of both operators is similar under the MA technique.

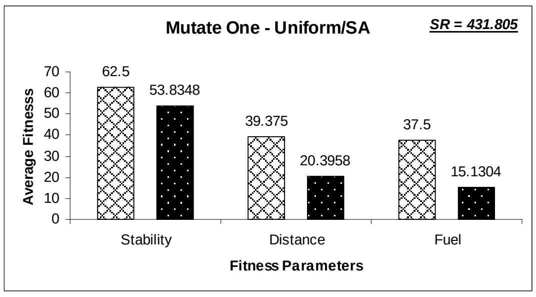
<u>Mutate One Results:</u>





(a) (b)



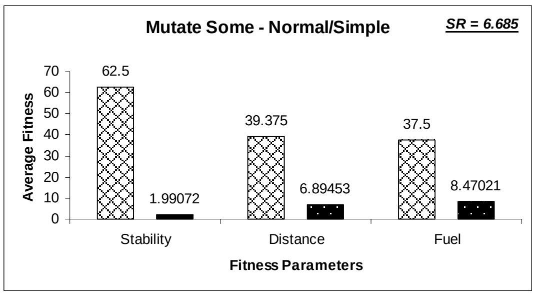


(c) (d)

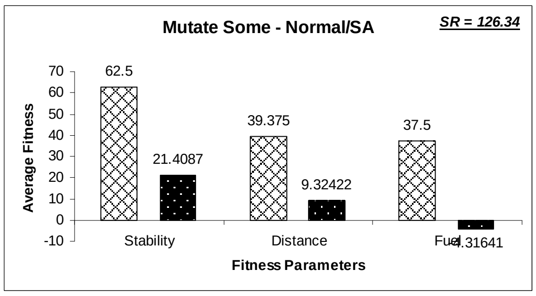
Figure 8.4 Normal Versus Uniform Distributions

Similarly to the MA technique, the Normal and Uniform Distributions have performed similarly. However, compared to the MA technique the MO's performance is slightly less. This is because the MO technique evolves one set of parameters (one neuron) close to the expected fitness, and then it moves to the next. Sometimes, it is likely to get stuck in one neuron and never find a way to evolve or move towards the expected fitness. This is a local minima situation and the SA is more effective than Simple acceptance using the MO technique. This comparison can be clearly seen from figure 8.4(a) & (c) – Simple acceptance and in figure 8.4(b) & (d) – SA acceptance.

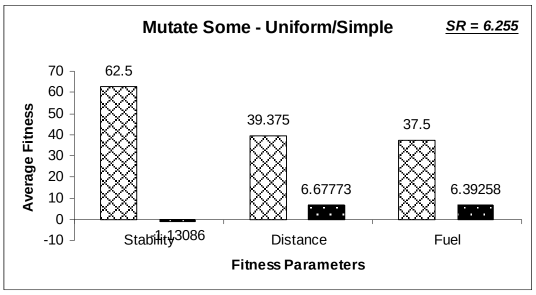
Mutate Some Results:



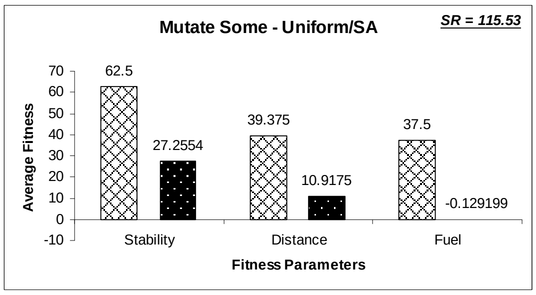
(a)



(b)



(c)



(d)

Figure 8.6 Normal versus Uniform Distributions

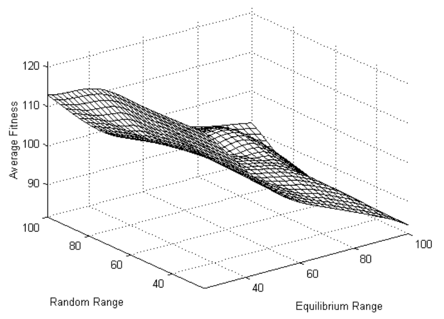
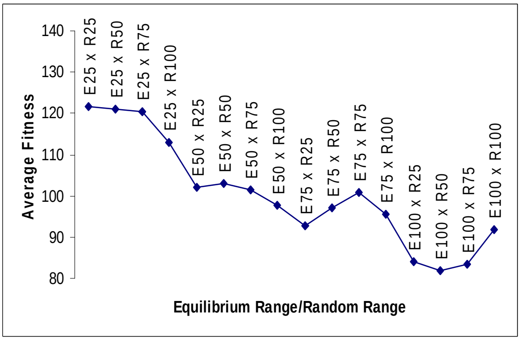
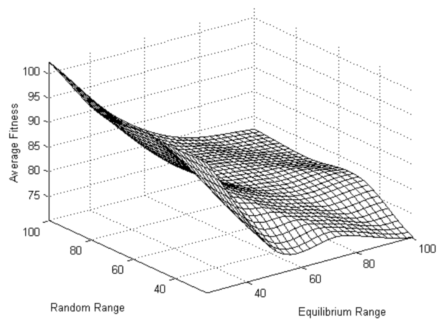
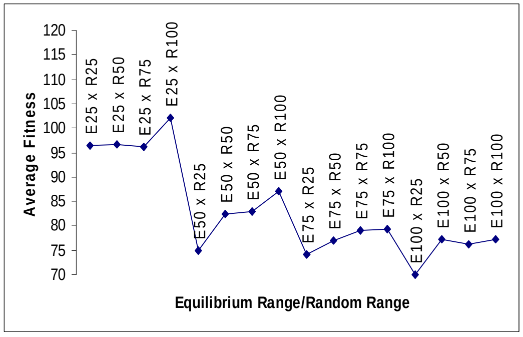
The Normal and Uniform distributions using the MS technique did not perform well because in all of the experiments all of the neurons are disturbed and it only evolves symmetrical neurons (front or back neurons – refer to section 4.2.2) at a time. As discussed earlier in the MA technique, when more than one neuron is mutated, in one cycle one set of neurons, for example front-***right***-leg neurons, may evolve close to expected fitness and in another cycle front-***left***-leg neurons may have caused the fitness to step-back, which will ultimately affect the fit front-***left***-leg neurons. This happens more under Simple acceptance than SA, as can be seen as the results in figure 8.5(b) and (d) are better than the results shown in figure 8.5(a) and (c).

8.3.2 Mutation Size versus Equilibrium Range

In this section, mutation size versus equilibrium range (ER) is discussed, to analyse their effects under Normal and Uniform distribution. Mutation size is also referred to as Random Range (RR) in this project. In this section, the graphs are shown in 3D and 2D views which may help in understanding the effects of ER and RR. Note that in the 3D graphs shown in this section, the ER axis and RR axis start from 25 and not zero. Each point on the graphs is an average of 256 experiments which covers the (System Disturbance) x (Stability) x (Distance) x (Fuel) = (4 x 4 x 4 x 4) (refer to table in figure 8.1). Also, for each point in the graph, the expected average fitness is ***139.375***.

In the 2D graph, each point is the average of the experiments done with a certain equilibrium range and random (mutation) range. Thus E25 x R25 is the average of all the experiments done with an equilibrium disturbance of 25% and mutation (random) range of 25%.

8.3.2.1 Normal Distribution with Simple Acceptance

	3D View	2D View
Mutate All		
Mutate One		

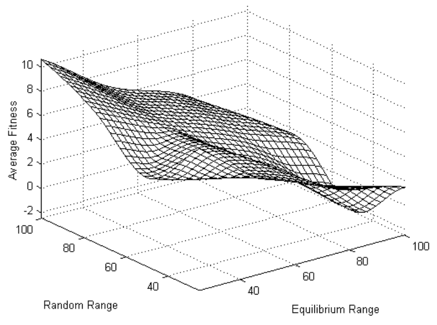
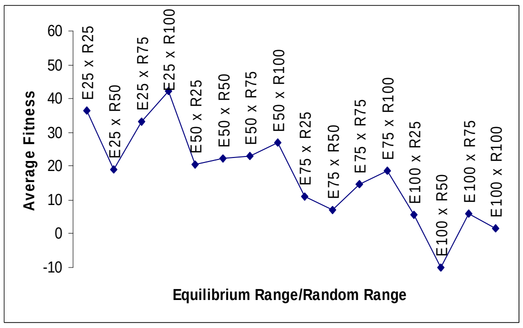
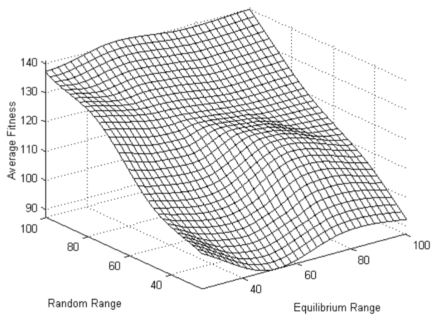
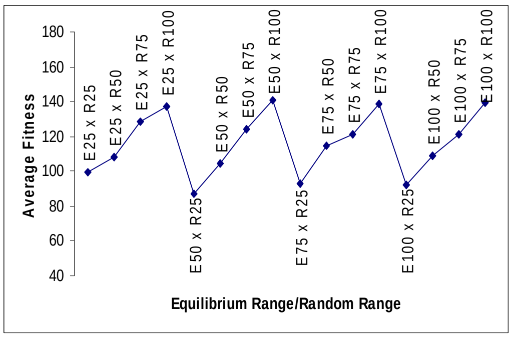
Mutate Some		

Figure 8.6 Equilibrium Range versus Random Range – Normal Distribution under Simple Acceptance

As the ER increases, the fitness decreases and for each ER (which goes from 25% to 100% in steps of 25) when RR increases, the fitness tends to increase, but does not get close to the expected fitness. This may be expected because it is obvious that, as the system moves away from equilibrium position, the poorer the fitness becomes. Also, as the mutation size or RR increases, the search space increases which increases the chance of improving the fitness. However, none of these examples evolves close to the expected fitness which is 139.375. This is because the “Simple” acceptance type is being used and it only accepts changes in parameters if the three fitness parameters (stability, distance and fuel) evolve close to the expected fitness. As there are three parameters to satisfy, the chance of getting close to the expected fitness is less and this is directly reflected in the graphs shown in figure 8.6.

8.3.2.2 Normal Distribution with Simulated Annealing

	3D View	2D View
Mutate All		

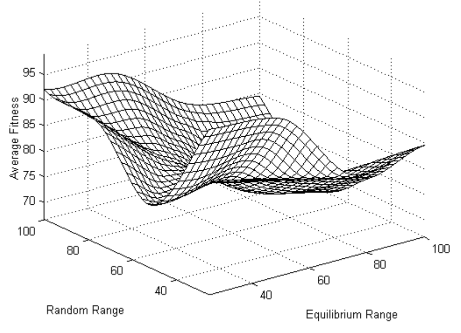
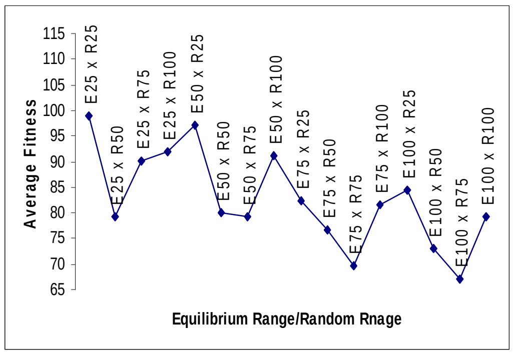
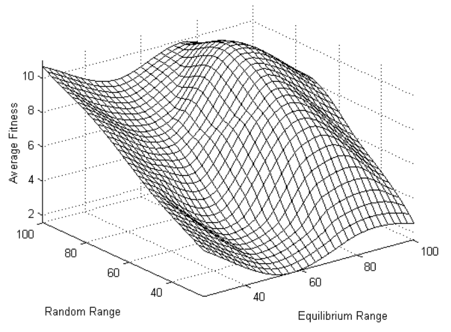
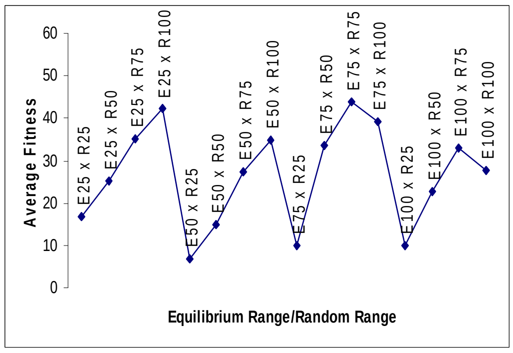
Mutate One		
Mutate Some		

Figure 8.7 Equilibrium Range versus Random Range – Normal Distribution under Simulated Annealing Acceptance

When the MA technique is used, the SA acceptance helps the system to improve fitness even when moving away from the equilibrium position. This can be seen in the graphs shown in figure 8.7 (top graph). It can be said here that, as the ER increases, the SA helps the fitness to increase (unlike Simple acceptance which decreases) and for each ER, when RR increases the fitness increases. However, this does not necessarily apply to the MO or MS technique because only one set of symmetrical parameters are mutated. For example, consider two neurons in a cycle where the SA has allowed an error; as a result of this, the stability has improved in one neuron but completely lost the fuel efficiency in another. In this case, the effect of ER and RR is difficult to analyse. Hence, the graphs do not show any symmetry effect similar to when MA is applied.

8.3.2.3 Uniform Distribution with Simple Acceptance

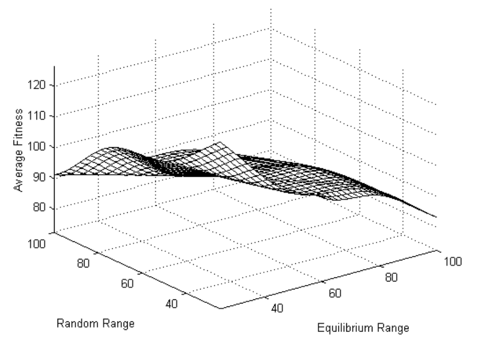
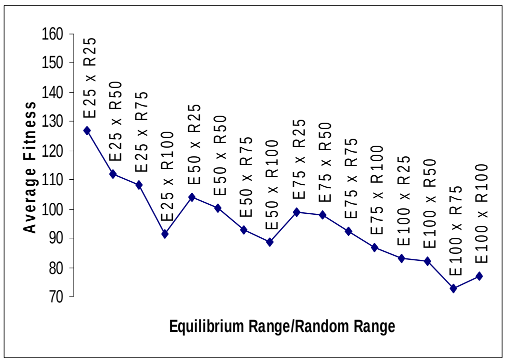
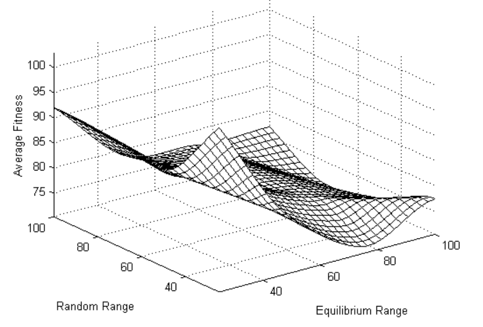
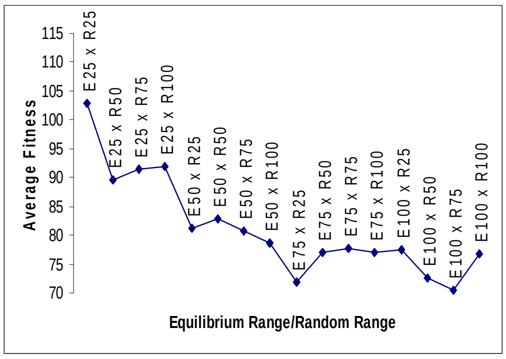
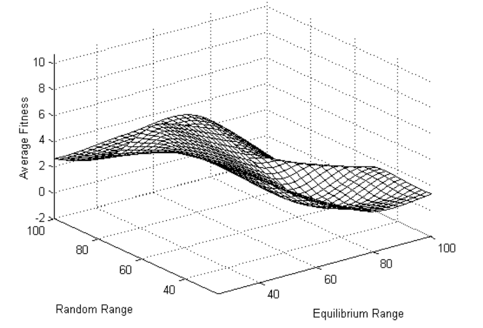
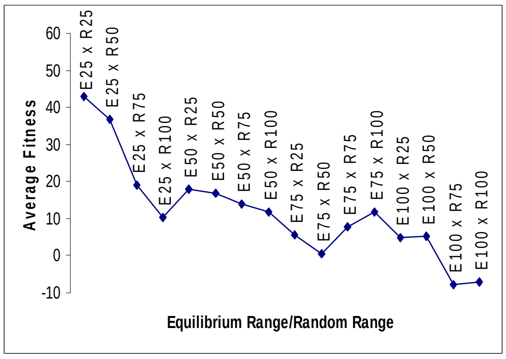
	3D View	2D View
Mutate All		
Mutate One		
Mutate Some		

Figure 8.8 Equilibrium Range versus Random Range – Uniform Distribution under Simple Acceptance

The effect of ER and RR (refer to figure 8.8) under Uniform Distribution/Simple Acceptance is close to the effect when Normal Distribution/Simple Acceptance is applied (refer to section 8.3.2.1).

8.3.2.4 Uniform Distribution with Simulated Annealing

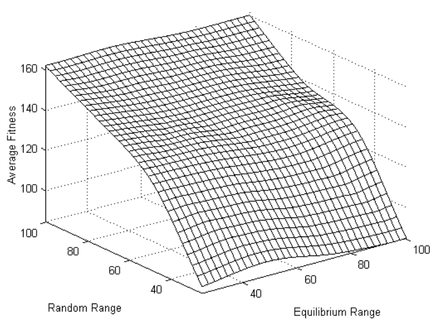
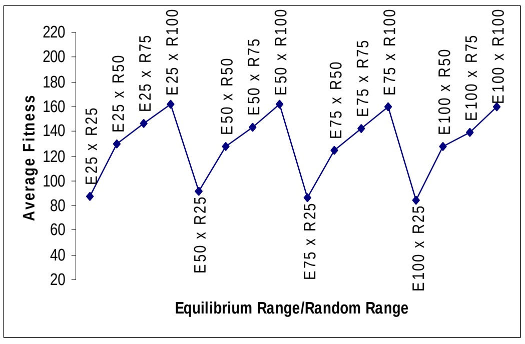
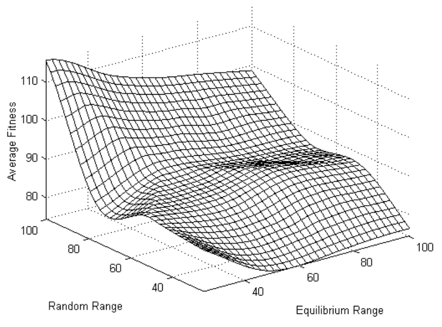
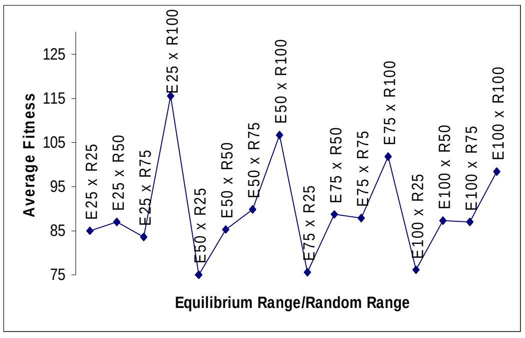
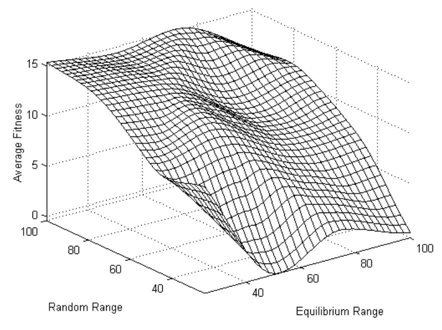
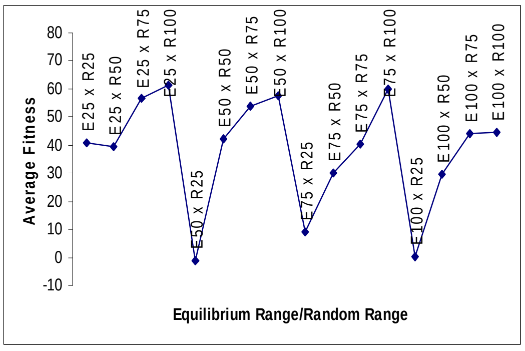
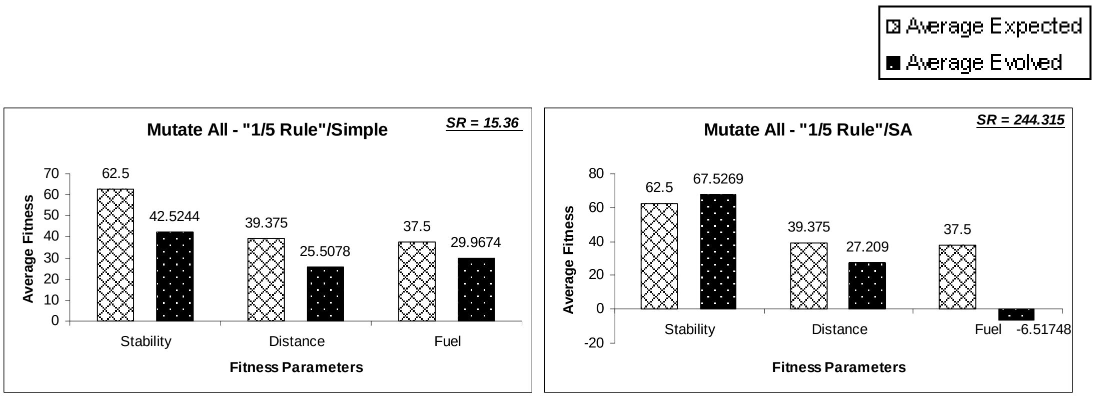
	3D View	2D View
Mutate All		
Mutate One		
Mutate Some		

Figure 8.9 Equilibrium Range versus Random Range – Uniform Distribution under Simulated Annealing Acceptance

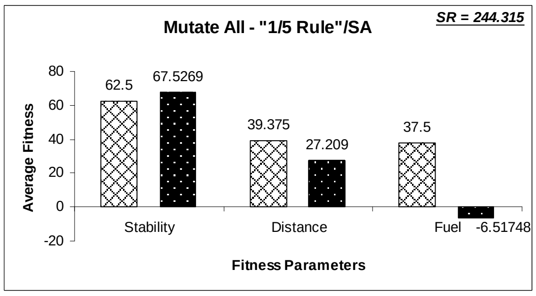
Similarly, the ER/RR effects (refer to figure 8.9) seen under Normal distribution were found again here under Uniform distribution. This confirms that both the operators give similar effects in real-time operations.

8.3.3 Implication of 1/5 Rule

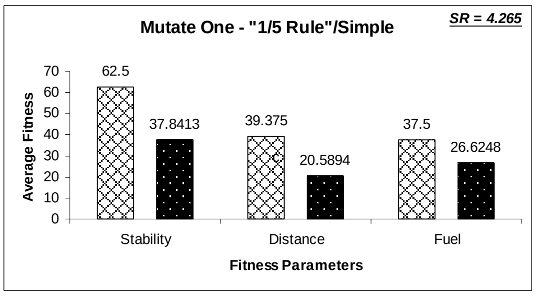
"1/5 Rule" operator is explained separately from the other two operators because it uses a slightly different experimental setup. In this case variations of mutation size are omitted so that the effect of increasing and decreasing the variance in the 1/5 Rule may be effectively studied. Thus the results (each graph) discussed here are an average of ***1024*** experiments. Apart from the number of experiments, the method of analysis is similar to method used with the other two operators. Refer to section 8.3.1.



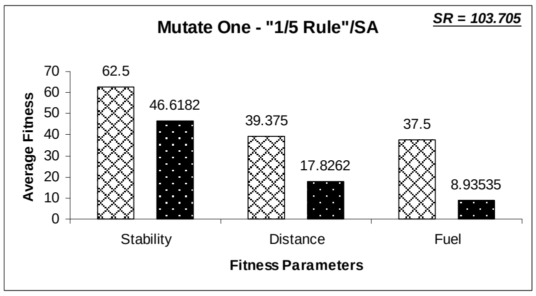
(a)



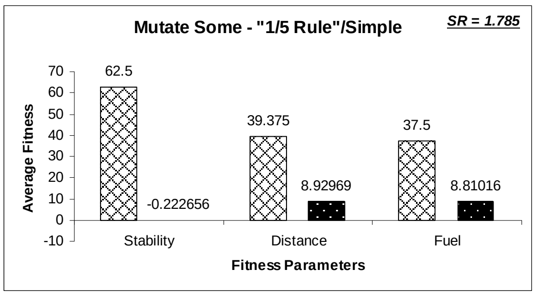
(b)



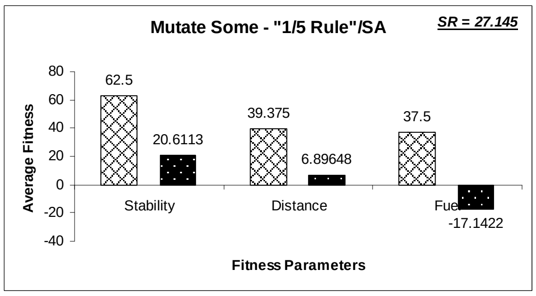
(c)



(d)



(e)



(f)

Figure 8.10 - 1/5 Rule

In figure 8.10, figure (a) and (b) are the results of "1/5 Rule" when the MA technique is applied with Simple and SA acceptance. Similarly, the results of the MO and MS techniques are shown in figure (c), (d), (e) and (f).

The "1/5 Rule" did not show any significant difference from the other two operators in this project. This is largely due to the fact that the system in which the operator is being tested shows changes made by whole numbers, as explained in section 8.3.1.

The results seen so far are summarised in bullet points in the next section. Further results are attached in appendix-B. A paper outlining these results was presented in an International Conference and a copy of this paper is attached in appendix-A.

8.4 Summary of RTEA Operators

- Normal/Simple/SA and Uniform/Simple/SA performed similarly under each mutation technique.
- MO with SA performed better than MA with SA. But the overall performance of MA was better than MO.
- MS performed poorly due to the combinational effect of symmetrical parameters mutating and disturbing all the neurons all of the time.
- 1/5 Rule did not show any significant importance in real-time operation; however, the effects were quite similar to the Normal and Uniform operators.

Chapter 9

Future Work

9.1 Introduction

This chapter suggests some future work that can be carried out after this project. The chapter starts by explaining some applications of RTEAs, other than gait control. Next, some investigations into network topology and gait transitions are suggested; these are backed up with some initial results. Finally, some suggestions are made for applying the RTEA to a real (physical) robot.

9.2 Other Applications of RTEA

In this section some other applications of the RTEA are explained. The Artificial Neural Network was originally chosen as a vehicle for exploring the RTEA because it may be applied to many different systems. For example, ANNs can be used not just in robotic systems but in many other control tasks.

The RTEA can also be used without the ANN in many tasks. For example, in the case of the robot, if the leg patterns were controlled by a simple software loop (switching on and off the motors at various times), then the RTEA could be applied directly to the timings of the leg movements. The paragraphs below give some examples (out of a great many possible) of the use of the technique.

One possible application is a combination with the earlier work by Muthuraman [1], within the research group. Muthuraman developed a successful method of evolving complex systems using a modular approach. However, the modules were trained off-line before use and could not change or optimise themselves in use. Combining the RTEA with this system may allow the system to optimise itself as it is being used, the module parameters and weights being the subject of the real-time evolution.

Another application may be in Aircraft Control Systems. There are several automatic control systems currently used in such applications as explained by McLean [2]. Some of these have turned to Artificial Intelligence for complex control tasks. For example, Faller and Schreck proposed a Neural Network solution for the prediction of unsteady Aerodynamic situations [3]. The application of a RTEA to optimise such systems could be investigated. This might be particularly interesting in situations where the aircraft is damaged or the control function is unknown (indeed, the group has already developed a method which may be used to find approximate minima in such situations [4]).

Robots themselves can and will play a key role in exploring space and other hazardous areas (an example being the recent NASA Mars Rovers). The current control systems for such vehicles are complex and involve several stages of analysis. This has to be done automatically since, as in the example of space exploration, the system is not in real-time contact with the operator. Obviously the RTEA is one technique which might benefit such systems is allowing them a method by which they can negotiate difficult terrain.

9.3 Network Topology

As the research progressed, it came to be understood that the network topology was one of the key factors in the real-time system. The simpler the network, the simpler is the control. After considering the functionality requirements of the network and the lessons learned from the earlier experiments, the network shown in figure 9.1 was found to be probably the simplest topology that would fulfil all of the requirements. This conclusion was arrived at by means of trial and error. Similar to the quadruped network shown in chapter 7, it also has mother and child neurons.

As may been seen in figure 9.1, the network has three layers of four neurons. The central layer contains the (clocking) mother-neurons. Each of the mother-neurons is connected to one vertical child-neuron (top layer) and one horizontal child-neuron (bottom layer). The weight from the mother-neuron to the child neuron depends on the actuator characteristics. For example, if the front-left-vertical child-neuron has to stretch its vertical motion by 2.5 units from its starting position, then the weight between the MN-LF and CN-V has to be 2.5 as shown in the figure.

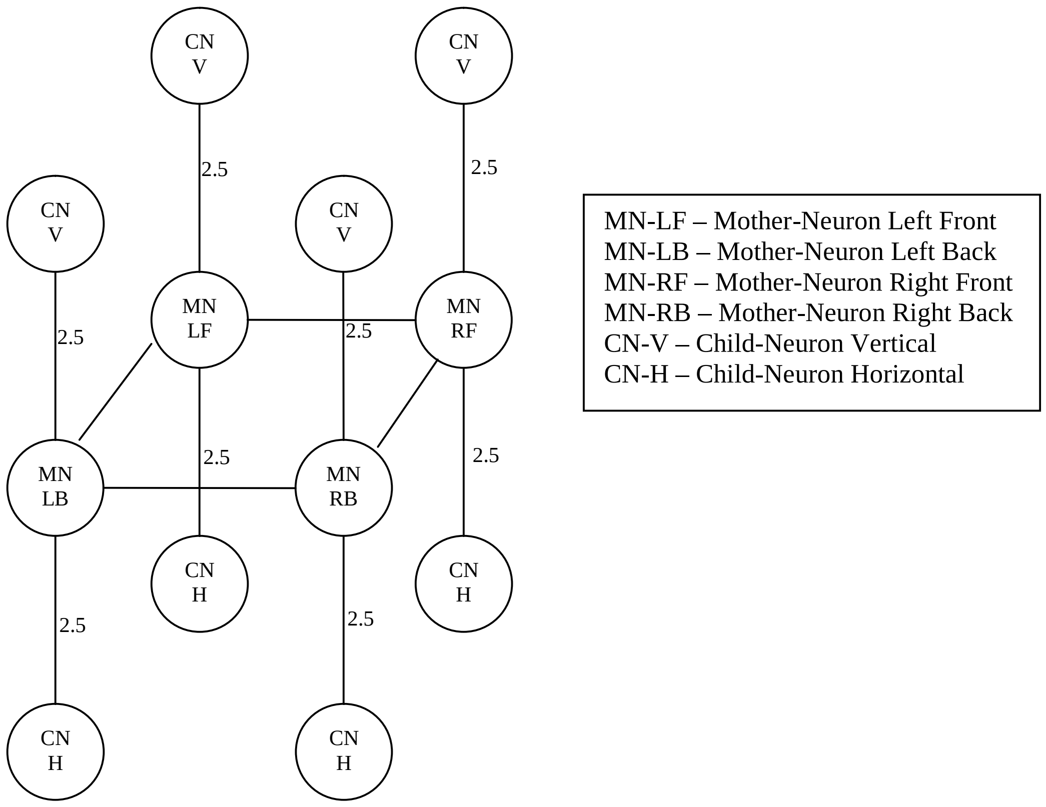

CN V
CN V
2.5
2.5
CN V
CN V
MN LF
2.5
MN RF
2.5
2.5
2.5
MN LB
MN RB
2.5
CN H
CN H
2.5
2.5
CN H
CN H
MN-LF – Mother-Neuron Left Front
MN-LB – Mother-Neuron Left Back
MN-RF – Mother-Neuron Right Front
MN-RB – Mother-Neuron Right Back
CN-V – Child-Neuron Vertical
CN-H – Child-Neuron Horizontal

Figure 9.1 Neural Network Topology for Quadruped Robot

The type of gait generated by the network is decided by an input tonic, similar to the input tonic shown by Billard and Ijspreet in their paper [5]. The input tonic for each gait is shown below

1	Walk	+1	-1	-1	-1
2	Gallop	+1	-1	-1	+1
3	Trot	+1	-1	+1	-1
4	Pace	+1	+1	-1	-1

The input tonic for each gait starts from the MN-RF to the MN-LF, going clockwise. For example, for walk gait, the input tonic is fed to the network as shown below.

MN-RF = +1

MN-RB = -1

MN-LB = -1

MN-LF = -1

In the investigation, the vertical child-neurons were initially kept passive and only the horizontal child-neurons were active. A child neuron is activated if it receives a positive pulse from its mother-neuron, with the condition that it had a negative pulse previously from the mother-neuron. The neuron activation function is given in equation 9.1.

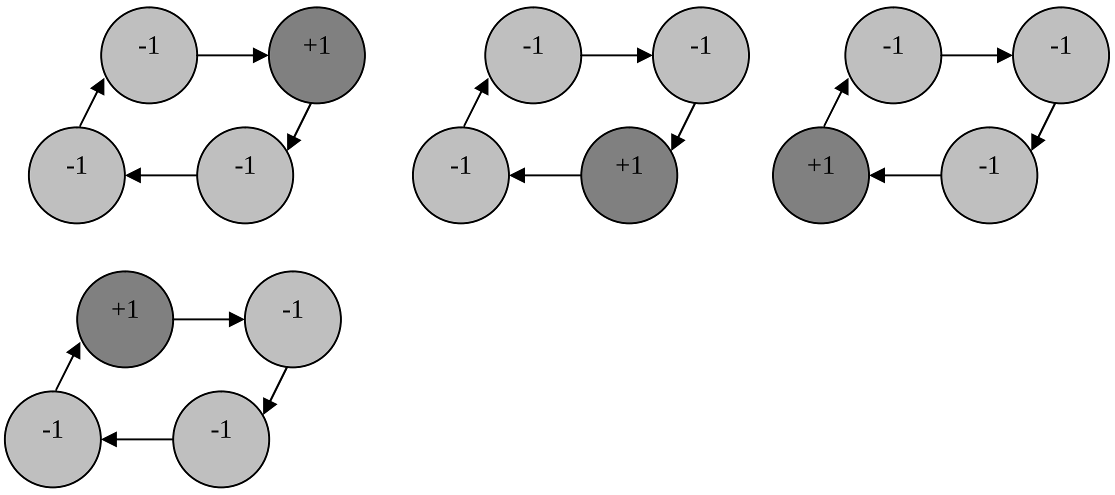
$$\begin{aligned}
&if\,((CN \xleftarrow{Receives} +1) and (CN \xleftarrow{Previous\_Signal} -1)) \\
&\Rightarrow Net = (Input \times Weight) \quad and \quad Output = \left(1 + \left(\frac{1}{1+e^{-Net}}\right)\right) \\
&if\,((CN \xleftarrow{Receives} +1) \quad and \quad (CN \xleftarrow{Previous\_Signal} +1)) \\
&\Rightarrow Net = (-1 \times Input \times Weight) \quad and \quad output = \left(1 + \left(\frac{1}{1+e^{-Net}}\right)\right) \\
&if\,((CN \xleftarrow{Receives} -1) \quad and \quad (CN \xleftarrow{Previous\_Signal} -1)) \\
&\Rightarrow NoAction
\end{aligned} \qquad (9.1)$$

Where CN is Child-Neuron

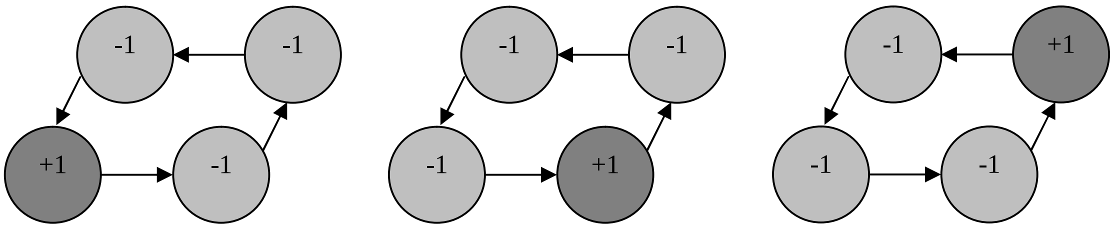
As can be seen from equation 9.1, if the neuron receives a positive pulse and the previous pulse received was a negative pulse, then it is activated as shown. If the neuron receives a positive pulse and the previous pulse received was also positive, then the neuron's previous action is reversed by multiplying by "-1" as shown in equation 9.1. If the neuron receives a negative pulse and the previous pulse received was also a negative pulse, then the neuron performs no action.

After activating the child neuron according to the input pattern, the mother-neurons "swing" the input pattern (that is, they pass each parameter to the next in line), so that, each mother-neuron gets its turn to activate its child-neuron. The swinging mechanism is shown in figure 9.2. First, the mother-neuron swings the input pattern clockwise, as shown in figure 9.2a and then it swings anti-clockwise as shown in figure 9.2b. The mother-neuron keeps this going as long as it is required.

The neural network's output was simulated on a quadruped robot (similar to the robot shown in chapter 7) which moves its actuators according to the input it receives from the child-neuron. For example, if the neuron sends 1.5 to the left-front-horizontal actuator, then the actuator moves to position 1.5. The position of each actuator of the robot is calibrated as shown in figure 9.3.



(a) Clockwise Swing



(b) Anti-clockwise Swing

Figure 9.2 “Swinging” Mechanism

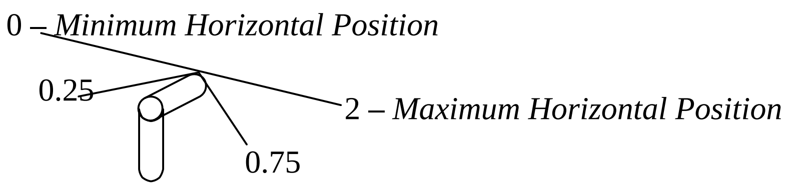

0 – *Minimum Horizontal Position*
0.25
2 – *Maximum Horizontal Position*
0.75

Figure 9.3 Robot Actuator Positions

The network and the robot were simulated to test the system. A screen-shot of one such simulation is shown in figure 9.4. The gait transition from walk to trot can be seen on the TChart in the screen-shot (figure 9.4), between 10 and 15 on the x-axis.

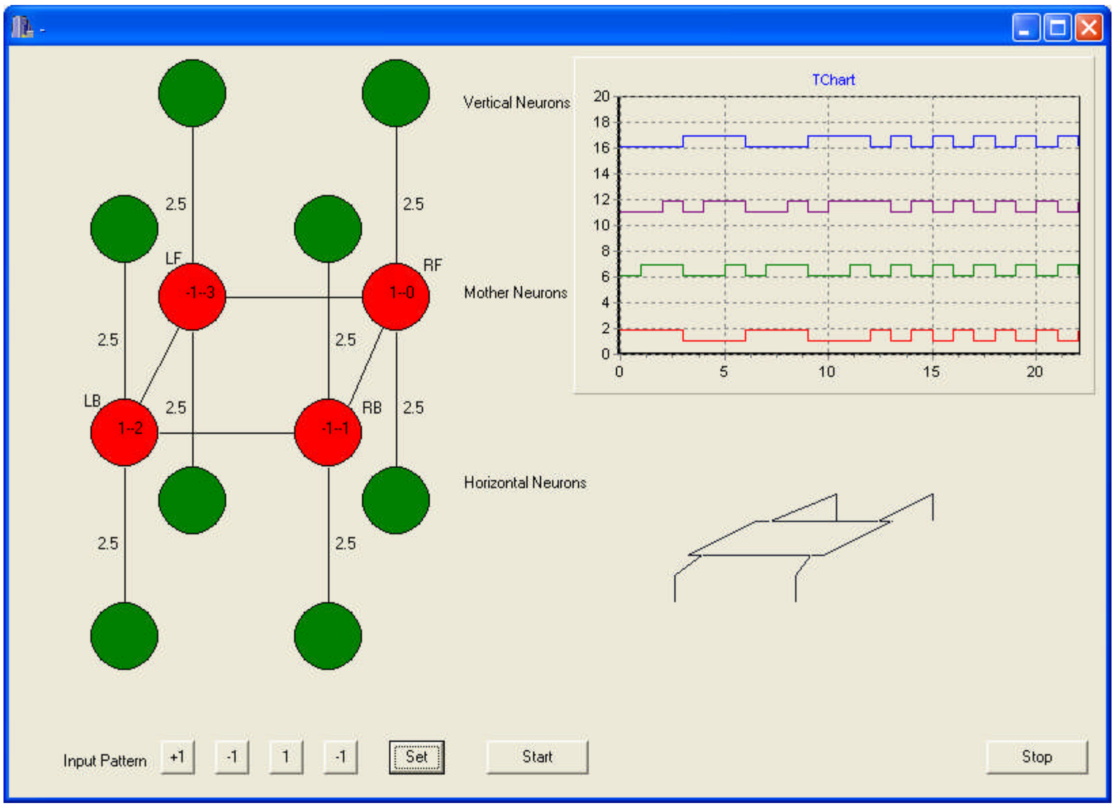


Figure 9.4 Gait Change Simulation

9.4 Changing Gaits

The neural network and the robot simulation showed good results and the RTEA was then applied to the system. Simulations were conducted with a “Mutate-All” and “Mutate-One” RTEA. The RTEA mutates the input tonic (patterns) to change the gait. The result of Mutate-All operation is shown in figure 9.5.

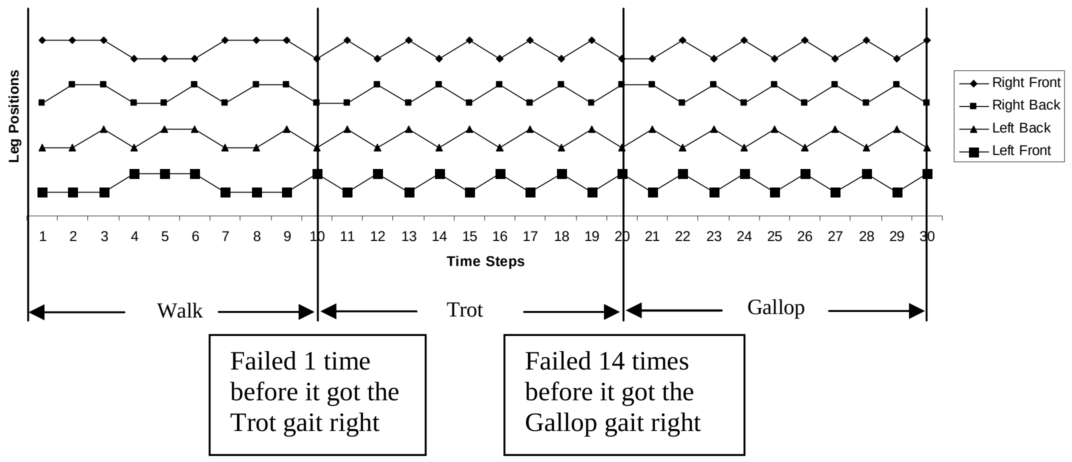


Figure 9.5 Result of application of Mutate All to gait changing operation

As the results show, the Mutate-All operation failed once before it changed the gait from walk to trot and failed 14 times before it changed from trot to gallop. The result of application of Mutate-One is shown in figure 9.6.

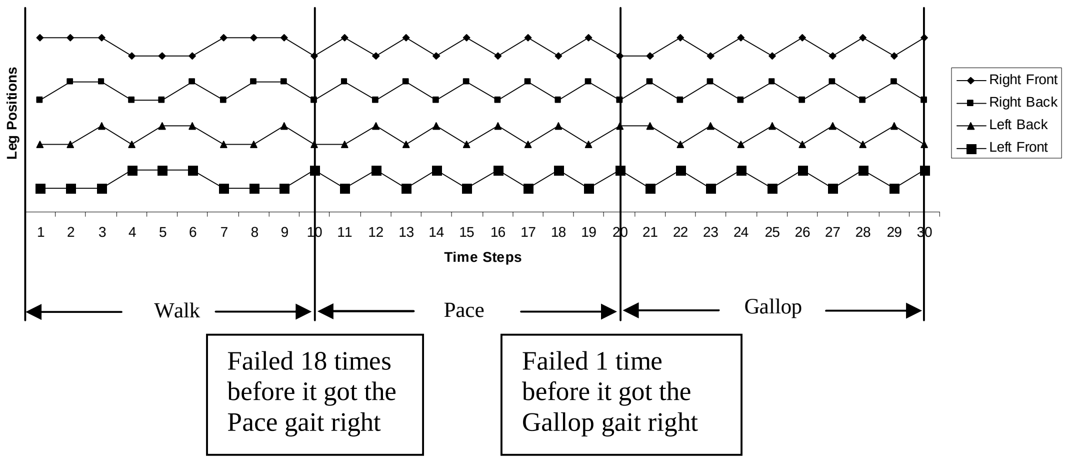


Figure 9.6 Result of application of Mutate One to gait changing operation

The results shows that the Mutate-One operation failed 18 times before it changed the gait from walk to pace and failed once before it changed the gait from pace to gallop. Further work can be done on this interesting system by including the following steps:

1. Include the vertical action of the robot by improving the network to get the vertical-child-neuron used.
2. Further work can also be done by not only mutating the input patterns, but also the weights, so that the calculation of efficiency of the RTEA may be more accurate.
3. The simulations can be improved by simulating rough terrain and letting the robot walk on it.

9.5 RTEA on Real Robot

All the experiments in this project were done in software simulation. In the future the RTEA can be investigated by applying it to a real-robot. The RTEA may be coded on a micro-controller board and the controller may be applied to the real quadruped robot (for example, as shown in picture 7.1 in chapter 7).

Chapter 10

Conclusion

10.1 Introduction

This final chapter presents the conclusions of the project. The chapter starts by reviewing the objectives which were set out at the beginning of the research. Then the original contributions of the research are presented. Next, a summary of future work is given. Finally, the chapter finishes by commenting on the overall success of the project.

10.2 Project Objectives Revisited

The project objectives, as originally stated at the beginning of the project, were:

1. Background Reading and Appropriate Directed Study
2. Literature Search in the Field
3. Development of a CPG ANN for a Bipedal Walking Robot
4. Investigation of Evolutionary Algorithms to train the CPG Network
5. Comparison with previously obtained results
6. Extension of MPhil work from Biped to Quadruped robot
7. Extension of the EA to a RTEA to train the CPG for a walking robot in any gait
8. Selection of best algorithm from the real time evolutionary algorithms developed
9. Running and testing the best algorithm
10. Comparison with published benchmarks and results from other researchers

The following sections look at each of these objectives in turn and explain how well they have been achieved.

10.2.1 Background Reading and Appropriate Directed Study

At the beginning of the research, appropriate background reading and study, required to understand the project, was undertaken. This was directed by the supervisors and included the coding and testing of practical ANNs and the examination of the previous work of the group [1]. This background is explained in detail in chapter 2.

10.2.2 Literature Search in Field

The study of relevant literature was undertaken throughout the research. Topics related to real-time neural networks, real-time robots and robotic control systems were the main areas covered. Chapter 5 explains the outcome of the literature search.

10.2.3 Development of a CPG ANN for a Bipedal Walking Robot

After examining the working principles of McMinn's [1] artificial neuron model, certain changes were made to eliminate its disadvantages and a new model was designed. This new model emphasised time properties, because the project involved handling real-time situations. The new artificial neuron has a similar behaviour to a biological neuron when fired. It produces positive pulses (ON cycle) and negative pulses (OFF cycle) for certain time steps. Also, when a neuron is fired, it will be ready for another excitation only after it completes its ON and OFF cycle. For a biped walking robot system, the network (the CPG) consisted of two neurons which were cross-connected. The output of each neuron was connected to a leg of the robot. Hence, when the neuron gave a positive pulse, the leg moved in a clockwise direction and when the neuron gave a negative pulse, the leg moved in an anti-clockwise direction. At the end of the work an artificial CPG for a Biped walking robot was developed. This is illustrated in chapter 6.

10.2.4 Investigation of EAs to train the CPG ANN

An initial implementation of the RTEA was developed to control the CPG of the biped robot. It was shown that a robotic neural control system can be trained and controlled by the algorithm. The results are outlined in chapter 6.

10.2.5 Comparison with previously obtained results

McMinn showed that time-dependent neuron models perform better than the McCulloch-Pitt's model. Instead of using McMinn's model, a similar time-dependent neuron model was developed which takes advantage of the RTEA's unique attributes. The result of this ANN and the other ANNs developed during the course of this project are shown in chapter 6 where the results were compared with each other. The differences in the models mean that it is difficult to compare the current work with the previous. However, it may be seen that the results are broadly similar.

10.2.6 Extension of MPhil work from Biped to Quadruped robot

The MPhil work was extended to PhD work by taking the steps below:

1. Development of biped robot system to quadruped robot system
2. Developing a simulator for the quadruped robot
3. Simulating and testing the quadruped robot system
4. Development of a Fitness-function for the quadruped robot
5. Development of a Neural Network for the quadruped
6. Development of a Network Topology for the quadruped
7. Simulating and testing different network topologies
8. Selecting the best topology

All the above steps were followed and discussed in detail in chapter 7 and the following chapters

10.2.7 Extension of Evolutionary Algorithm to RTEA to train the CPG ANN for a walking robot in any gait

The general RTEA method is shown in basic form (initial tests were done simply to get an idea of the system dynamics) in chapter 4. The RTEA was initially tested on the biped. After observing the performance of the RTEA on the biped, different RTEAs for the quadruped robot were developed. This is discussed in chapter 8.

10.2.8 Selection (and running) of best algorithm from the RTEAs developed

Different RTEAs were tested as shown below.

1. Mutate All
2. Mutate One
3. Mutate Some

These RTEA operators are explained in chapter 4. Different “Random Number Distributions” and “Conditions of Acceptance” were used to test the RTEAs and they are also explained in chapter 4.

Chapter 8 examined the different experimental parameters by which the RTEAs may control the robot. Finally, the results of different RTEAs and their different operators were also discussed and shown in chapter 8. A summary of the results is given below.

- Normal/Simple/SA and Uniform/Simple/SA performed similarly under each mutation technique.
- MO with SA performed better than MA with SA, but the overall performance of MA was better than MO.
- MS performed poorly due to the combinational effect of symmetrical parameters.
- The 1/5 Rule did not show any significant importance in real-time operation and the effects were quite similar to the Normal and Uniform operators.

It should be noted that the experimental objectives moved away from finding “the best” algorithm to finding the effects of different operators. This was due to recognition of the complexity of the problem and the fact that different operators may be beneficial in different situations; there may not be a “best” overall system setup.

10.2.10 Comparison with published benchmarks and results from other researchers

As explained in the literature review in chapter 5 there are no directly comparable systems to be found in the literature. This was not understood (or expected) at the start of the project. Therefore, a statistical comparison of different techniques is not available. However, it is obvious looking at real-time robotic control systems, that the RTEA performs similarly to other pseudo random methods (like GAs, ES, etc.) but not as well as programmed systems (like subsumption architecture). However, the programmed systems lack flexibility as described previously.

10.3 Summary of Main Findings

The main findings of the research are summarised in the points below.

- The neuron model is critical to success and certain components of traditional models, for example thresholds, cause problems. This is mainly because they do not allow small changes in the model's parameters to affect the fitness function and hence cause smooth evolution.
- Likewise, network structure and topology is also important. Fully connected networks in this case allow too much interference and interaction between different parts of the same network.
- RTEAs can be successful in ANN applications. The operators used depend on the nature of the problem to which they are applied.
- The technique is useful for instigating small refinements to the system, but in applications with a large search space, a more successful technique may be to combine it with pre-programmed minima (for example, in the case of a robot, pre-programmed gaits).

10.4 Original Contribution

The main areas in which the research outlined in this thesis is unique and its contributions to 'the art' are summarised below.

- The investigation of neural unit models for real-time evolutionary applications. The identification of problems with existing models and the development of new models which mitigate these problems.
- A similar investigation as above, but for network topologies.
- The integration of these points into a working simulation.
- The testing and categorisation of evolutionary operators in real-time situations (in 18,462 different scenarios) and their development into working RTEAs.
- The development and testing of fitness functions for such situations.

10.5 Summary of Further Work

The future work suggested in chapter 9 may be summarised as follows.

- An investigation into the application of the RTEA in Muthuraman's work [2].

- An investigation of the application of the RTEAs to ANNs and similar systems in other applications.
- Further investigation of the gait changing problem and other network topologies.
- The application of the RTEA to a real robot.

10.5 Concluding Remarks

This project has been very successful in investigating the use of RTEAs in robotic systems and the RTEA has been shown to be a powerful technique for real-time operation.

The author therefore feels that the project is a useful contribution to the field of real-time ANN control systems. The simplicity of operation of the RTEA may prove useful in control system problems in the future.

References

Chapter 1

1. Nielsen, H. (1990) Neurocomputing, 1st ed. Reading, Massachusetts: Addison-Wesley Pub. Co, pp.1-20.
2. Antsaklis, P.J. (1990) Neural Networks for Control Systems, IEEE Transactions on Neural Networks, 1(2), pp.242-4.
3. Carpenter, G. A. and Grossberg, S. (1987) ART - 2: self organisation of stable category recognition codes for analog input patterns, Applied Optics, 26, pp. 4919-30,

Chapter 2

1. Wynsberghe, D.V., Noback, C.R. and Carola, R. (1995) Human Anatomy and Physiology, 3rd ed. New York: McGraw-Hill.
2. Bordal, P. (1992) The Central Nervous System: Structure and function, 1st ed. New York: Oxford University Press, pp. xiii (Introduction).
3. Levitan, I.B. and Kaczmarek, L.K (1997) The Neuron: Cell and Molecular Biology, 2nd ed. New York: Oxford University Press, pp. 6.
4. Giese, K.P., Peters, M. and Vernon, J. (2001) Modulation of excitability as a learning and memory mechanism: A molecular genetic perspective, Physiology and Behaviour, 73, pp. 804.
5. McMinn, D. (2002) Using Evolutionary Artificial Neural Networks to Design Hierarchical Animat Nervous System, PhD Thesis, The Robert Gordon University.
6. Bordal, P. (1992) The Central Nervous System: Structure and function, 1st ed. New York: Oxford University Press, pp. 209-10.
7. Grillner, S. Wallén, P. (1985) Central Pattern Generators for Locomotion, with special reference to vertebrates, Annual Review of Neuroscience, 8, pp. 233-61.
8. MacLeod, C., Maxwell, G.M. and McMinn, D. (1998) A Framework for Evolution of an Animat Nervous System, EUREL European Advanced Robotics Systems Development: Mobile Robotics, Leiria, Portugal.
9. Reddipogu, A., Maxwell, G., and MacLeod, C. (2002) An Innovative Neural Network Based on The Toad’s Visual System, ACIVS 2002 (Advanced Concepts for Intelligent Vision Systems), Ghent, Belgium.

Chapter 3

1. Haupt, R.L. and Haupt, S.E. (2004) Practical Genetic Algorithm, 2nd ed. Wiley-Interscience, pp. 9.
2. Grant, V. (1985) The Evolutionary Processes, New York: Columbia University Press.
3. Tagliaferrol, L. and Bloom, M.V. (1999) The Complete Idiot's Guide to: Decoding your Genes. Alpha Books, pp.73-75.
4. Browne, J. ed. and Neve, M. ed. (1839) Voyage of the Beagle: Charles Darwin's Journal of Researches, Penguin Books.
5. Biology Online Team (2001) Biology Online (Online). Available from: http://www.biology-online.org/2/10_natural_selection.htm (Accessed 21 Oct. 2005).
6. Goldberg, D.E. (1989) Genetic Algorithm in Search, Optimization and Machine Learning, Reading Massachusetts: Addison-Wesley Pub. Co, pp. 1.
7. Haupt, R.L. and Haupt, S.E. (2004) Practical Genetic Algorithm, 2nd ed. Wiley-Interscience, pp.32.
8. Eiben, A.E. and Smith, J.E. (2003) Introduction to Evolutionary Computing, London: Springer.
9. Haupt, R.L. and Haupt, S.E. (2004) Practical Genetic Algorithm, 2nd ed. Wiley-Interscience, pp.38-41.
10. Goldberg, D.E. (1989) Genetic Algorithm in Search, Optimization and Machine Learning, Reading Massachusetts: Addison-Wesley Pub. Co, pp. 11-12.
11. Goldberg, D.E. and Deb, K. (1991) A comparative analysis of selection schemes used in genetic algorithms: In Foundations of Genetic Algorithms. Los Altos, Morgan Kaufmann Pub., pp.69–93.
12. Poon, P.W. and Carter, J.N. (1995) Genetic Algorithms Crossover Operator for Ordering applications. Computers & Operations Research 22(1), pp. 135-147.
13. Haupt, R.L. and Haupt, S.E. (2004) Practical Genetic Algorithm, 2nd ed. Wiley-Interscience, pp.56-60.
14. Davis, L. (1987) Genetic Algorithm and Simulated Annealing. London: Pitman Pub., pp. 8.
15. Kirkpatrick, S., Gelatt Jr., C.D. and Vecchi, M.P. (1983) Optimisation By Simulated Annealing. Science, 221, pp. 671-680.

16. Weisstein, E.W. Normal Distribution. From MathWorld--A Wolfram Web Resource (Online). Available from: http://mathworld.wolfram.com/NormalDistribution.html (Accessed 21 Oct. 2005).
17. Manikas, T.W and Cain, J.T. (1996) Genetic Algorithm Ve. Simulated Annealing: A comparison of Approach for Solving the Circuit Partitioning Problem. Tech. Report TR-96-101, University of Pittsburgh, Dept. of Electrical Engineering.
18. Hasan, M., Alkhamis, T. and Ali, J. (2000) A Comparison between Simulated Annealing, Genetic Algorithm and Tabu Search Methods for the Unconstrained Quadratic Pseudo-Boolean function. Computer & Industrial Engineering, 38, pp. 323-340.
19. Lacksonen, T. (2001) Empirical Comparison of Search Algorithm for Discrete Event Simulation. Computers & Industrial Engineering, 40, pp. 133-148.
20. Kohonen, J. (1999) A Brief Comaprison of Simulated Annealing and Genetic Algorithm Approaches. Available from: http://www.cs.helsinki.fi/u/kohonen/papers/gasa.html (Accessed 19 Nov 2005).
21. Rechenberg, I. and Holzboog, F. (1973) Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution.
22. Schwefel, H.P. (1995) Evolution and Optimum Seeking, Wiley.
23. Bäck, T. and Schwefel, H.P. (1995) Evolution strategies I: Variants and their computational implementation. In Genetic Algorithms in Engineering and Computer Science, Proc. First Short Course EUROGEN-95. pp.111-126.
24. Fogel, L.J. and *et al*. (1966) Artificial Intelligence through Simulated Evolution, Wiley.
25. Chan, F.T.S., Chung, S.H. and Wadhwa, S. (2005) A Hybrid Genetic Algorithm for Production and Distribution, Omega: The International Journal of Management Science, 33, pp. 345-355.
26. Wang, H.F. and Wu, K.Y. (2004) Hybrid Genetic Algorithm for Optimization Problems with Permutation Property, Computer & operation Research, 31, pp. 2453-2471.
27. Bäck, T. (1996) Evolutionary Algorithms in Theory and Practice, New York: Oxford University Press, pp.73-78.

Chapter 5

1. Nehmzow, U. (1999) Mobile Robotics: A Practical Introduction, New York: Springer-Verlag, pp. 7-8.
2. Zeldman, M.I. (1984) What Every Engineer Should Know About Robots, New York: Marcel Dekker, pp. 3-12.
3. Brooks, R. A. (1986) A Robust Layered Control System for a Mobile Robot, IEEE Journal of Robotics and Automation, 2(1), pp. 14-23.
4. Connell, J. H. (1998) A Behaviour Based Arm Controller, MIT AI Laboratory Memo 1025, http://www.ai.mit.edu/research/publications/publications.shtml.
5. Rosenblatt, J. K. and Payton, D. W. (1989) A fine-grained alternative to the Subsumption Architecture for Mobile Robot Control, Proceedings IEEE/INNS International Joint Conference on Neural Networks, 2, pp. 317-324.
6. Maes, P. (1992) Learning behaviour networks from experience, Proceedings of 1st European Conference on Artificial Life (ECAL91), pp. 48-57.
7. de Garis, H. (1991) Building Artificial Nervous Systems Using Genetically Programmed Neural Network Modules, Parallel Problem Solving from Nature, Lecture Notes in Computer Science 496, Springer-Verlag.
8. de Garis, H. (1991) The Lizzy Project: Genetically Programming an Artificial Nervous System, International Conference on Artificial Neural Networks, Espoo, Finland.
9. de Garis, H. (1990) Genetic Programming: Building Artificial Nervous Systems Using Genetically Programmed Neural Network Modules, Proceedings of 7th International Conference on Machine Learning, Morgan Kaufmann, pp 132-139.
10. Gat, E. (1998) Three Layer Architectures, In Kortenkamp, D., Bonasso, R. P., Murphy, R. (eds), Artificial Intelligence and Mobile Robots, pp. 195-210.
11. Saridis, G. N. (1992) Architectures for Intelligent Control, Proceedings of International Symposium on Implicit and Non-linear Systems (SINS92).
12. Bartolini, G., Cannata, G., Casalino, G. and Ferrera, A. (1995) A Hierarchical Control Architecture for the Control of Underwater Robots, Proceedings of IFAC Workshop on Control Applications in Marine Systems, 3, pp. 60-65.
13. Sousa, J. B., Perera, F. L., da Silva, E. P., Martins, A., Matos, A., Almeida, J., Cruz, N., Tunes, R. and Cunha, S. (1996) On the Design and Implementation of a

Control Architecture for a Mobile Robotic System, Proceedings of IEEE International Conference on Robotics and Automation, 3, pp. 2822-2827.

14. Maurer, M. and Dickmanns, E. D. (1997) An Advanced Control Architecture for Autonomous Vehicles, Proceedings of SPIE , 3087, pp. 94-105.
15. Herbert, T., Valavanis, K.and Kolluru, R. (1998) A Real-Time Hierarchical Sensor-Based Robotic System Architecture, Journal of Intelligent and Robotic Systems, 21, pp. 1-27.
16. Digney, B. L. (1997) Learning and Shaping of Hierarchical Control Structures, Proceedings of 7th Topical Meeting on Robotics and Remote Systems, 1, pp. 30-37.
17. Yang, S.X. and Meng, M.D.-H. (2003) Real-time collision-free motion planning of mobile robots using neural dynamics based approaches, IEEE Transactions on Neural Networks. 14, Nov 6, pp. 1541-1552.
18. Vadakkepat, P., Tan, K.C. and Ming-Liang ,W. (2000) Evolutionary Artificial Potential Fields and their Application in Real Time Path Planning, IEEE International Conference on Evolutionary Computation, pp. 256-263.
19. Capi, G., Nasu, Y., Barolli, L. and Mitobe, K. (2001) Real Time Generation of Humanoid Robot Optimal Gait for going Upstairs using Intelligence Algorithm, Industrial Robot: An International Journal, 26(6), pp. 489-497.
20. Patńo, D.H., Carelli, R. and Kuchen, B.R. (2002) Neural Networks for Advanced Control of Robot Manipulators, IEEE Transactions on Neural Networks, 13(2), pp. 343-354.
21. Billard, A. and Ijspreet, A.K. (2000) Biologically inspired neural controller for motor control in a quadruped robot, Proceedings of the Fourth International Conference on Autonomous Agents.

Chapter 6

1. Sony Contact Centre Europe, (2004) Information of AIBO research. Available from: http://www.eu.aibo.com/5_3_research.asp (Accessed 15th Jan 2006).
2. McMinn, D. (2002) Using Evolutionary Artificial Neural Networks to Design Hierarchical Animat Nervous System, PhD Thesis, The Robert Gordon University.
3. Muthuraman, S. (2005), The Evolution of Modular Artificial Neural Networks, PhD Thesis, The Robert Gordon University.
4. McMinn, D., Macleod, C. and Maxwell, G. M. (2000) An Evolutinary Artificial Nervous System for Animat Locomotion, Sixth International Conference on Engineering Application of Neural Networks, Kingston Upon Thames, p 144-149.
5. McMinn, D., Macleod, C. and Maxwell, G. M. (2002) Evolutinary Artificial Neural Networks for Quadruped Locomotion, ICANN 02, Madrid, p 789-794.
6. Muthuraman, S., Maxwell, G. M., and Macleod, C. (2003) The Evolution of Modular Artificial Neural Networks for Legged Robot Control, Artificial Neural Networks and Neural Information Processing, Berlin, p 488-495.

Chapter 9

1. Muthuraman, S. (2005), The Evolution of Modular Artificial Neural Networks, PhD Thesis, The Robert Gordon University.
2. Mclean, D. (1990) Automatic Flight Control Systems, Prentice-Hall International.
3. Jet Propulsion Laboratory, California Institute of Technology. Available from http://marsrovers.jpl.nasa.gov/technology/is_autonomous_mobility.html (Accessed 17th May 2006)
4. Viswanathan, A., MacLeod, C., Maxwell, G. and Kalidindi, S. (2005) Training Neural Networks using Taguchi Methods: Overcoming Interaction Problems, ICANN 05, Warsaw Poland, Part II, p 103 – 108.
5. Billard, A. and Ijspreet, A.K. (2000) Biologically inspired neural controller for motor control in a quadruped robot, Proceedings of the Fourth International Conference on Autonomous Agents.

Chapter 10

1. McMinn, D. (2002) Using Evolutionary Artificial Neural Networks to Design Hierarchical Animat Nervous System, PhD Thesis, The Robert Gordon University.
2. Muthuraman, S. (2005), The Evolution of Modular Artificial Neural Networks, PhD Thesis, The Robert Gordon University.

Appendix

Appendix A

Paper Published during the PhD

Published in the proceedings of the International Conference of Neural Networks:

A Jagadeesan, G Maxwell, C MacLeod, "*Evolutionary Algorithms for Real-Time Artificial Neural Network Training*", ICANN 2005, Warsaw, Poland, Part II, p 73-78.

Abstract:

This paper reports on experiments investigating the use of Evolutionary Algorithms to train Artificial Neural Networks in real time. A simulated legged mobile robot was used as a test bed in the experiments. Since the algorithm is designed to be used with a physical robot, the population size was one and the recombination operator was not used. The algorithm is therefore rather similar to the original Evolutionary Strategies concept. The idea is that such an algorithm could eventually be used to alter the locomotive performance of the robot on different terrain types. Results are presented showing the effect of various algorithm parameters on system performance.

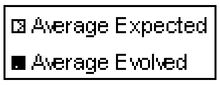
Note: This paper is not available in the electronic version of the thesis. It may be obtained from the publisher.

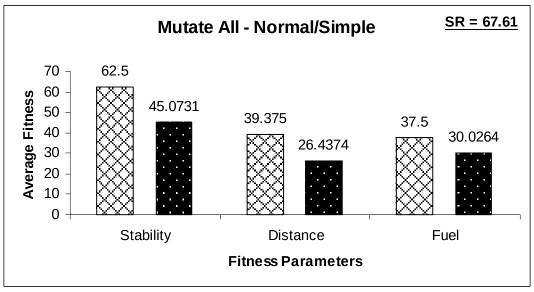
Appendix B
Further Results

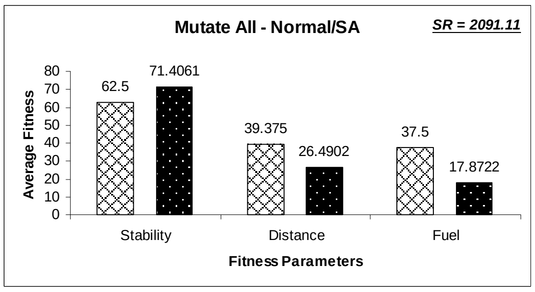
The graphs shown below are the result of simulating each experiment for 2000 cycles (unlike the results shown in chapter 8, which use 1000 cycles)

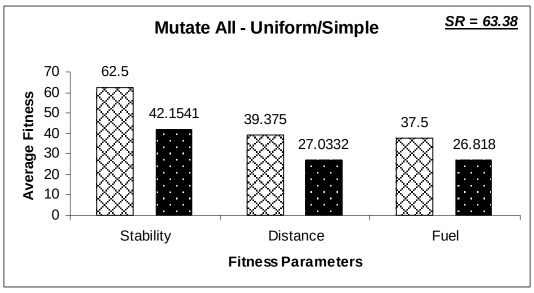
Normal Versus Uniform Distributions

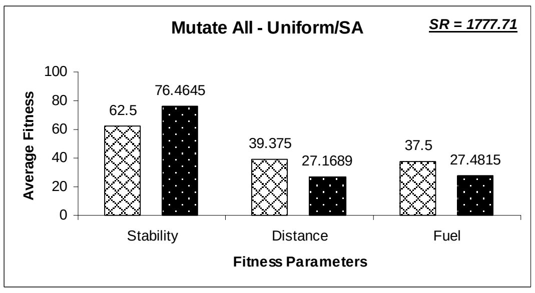
Mutate All Result:



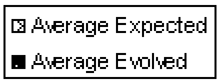


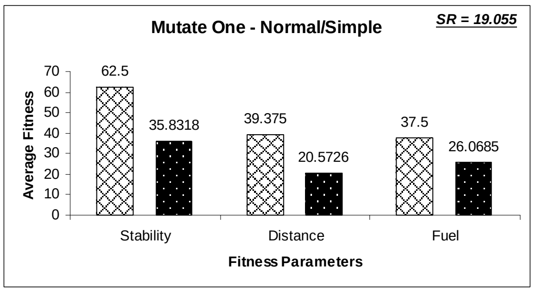


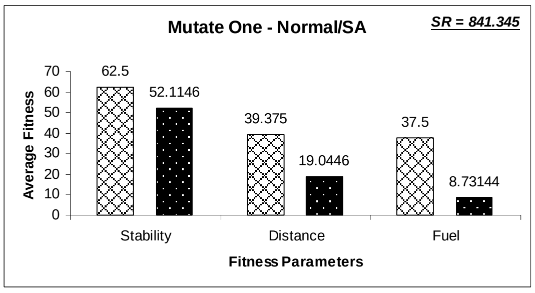


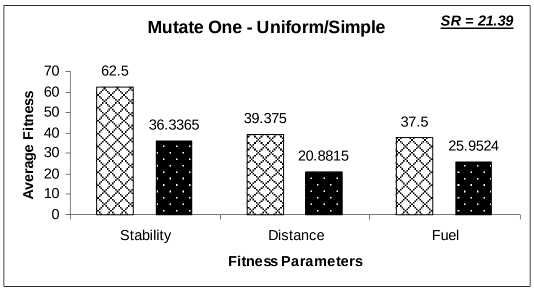


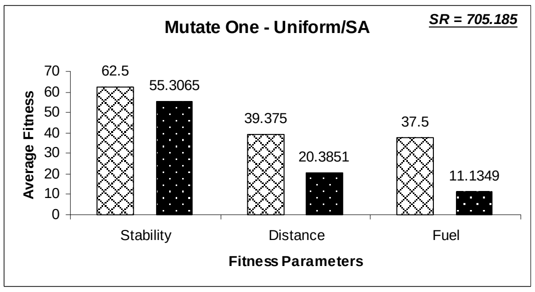
Mutate One Result:



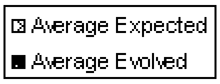


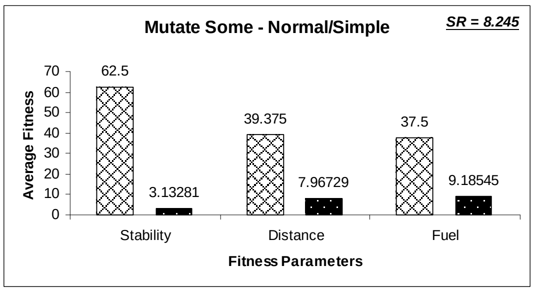


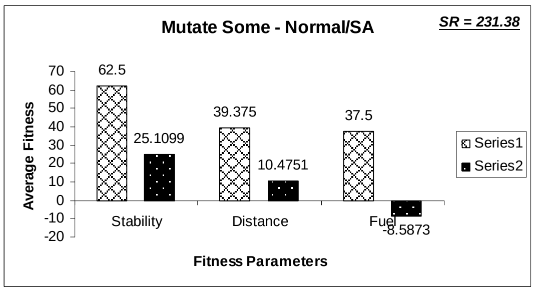


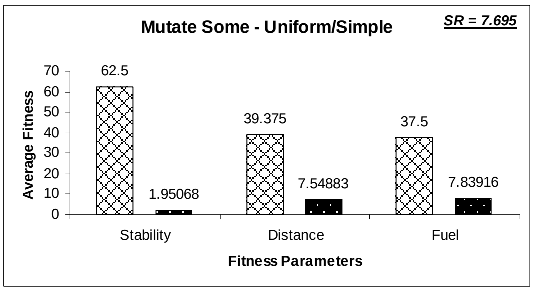


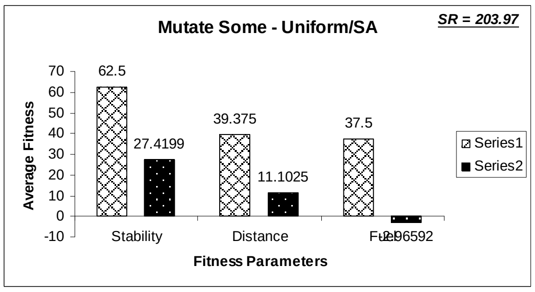
Mutate Some Result:





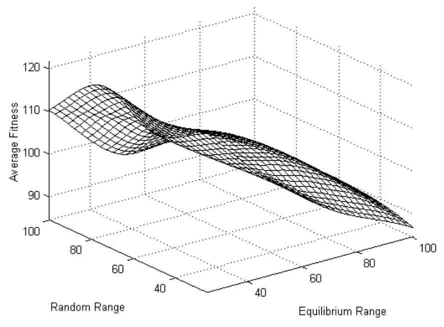
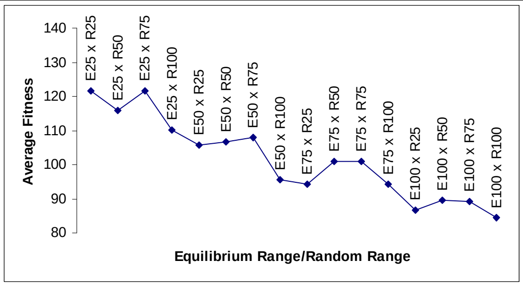
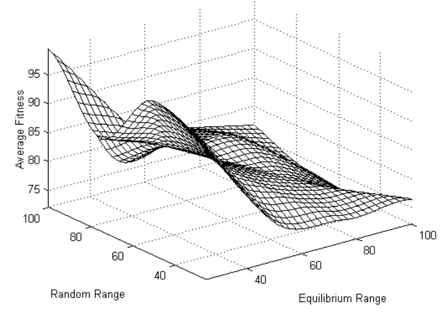
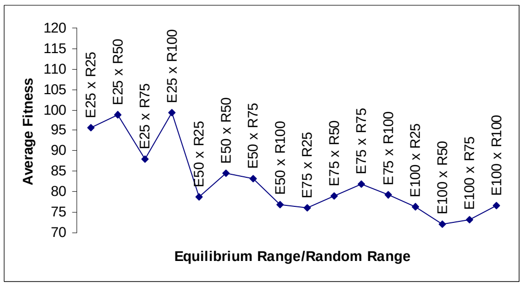
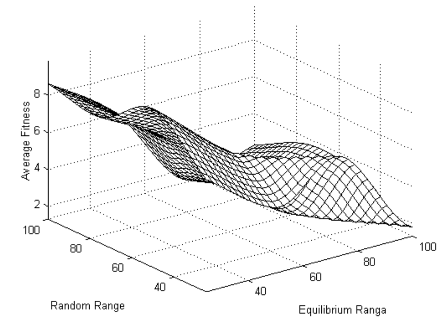
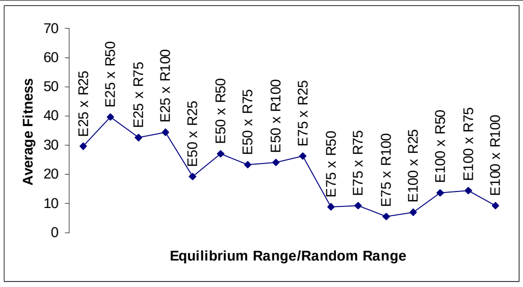




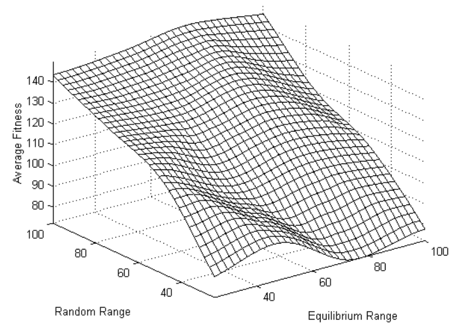
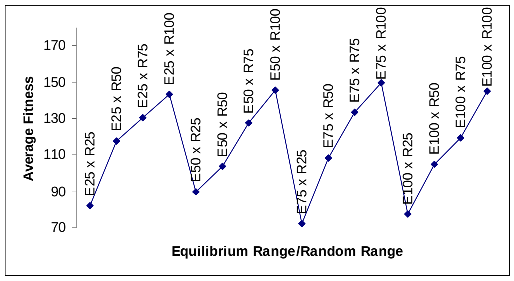
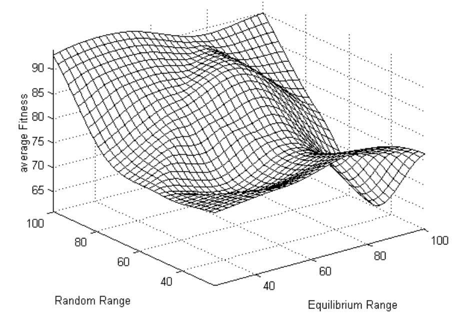
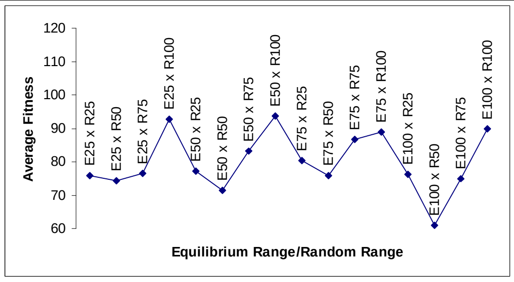
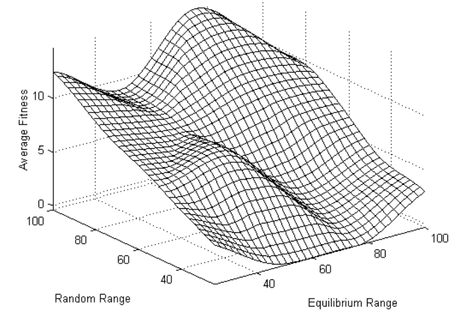
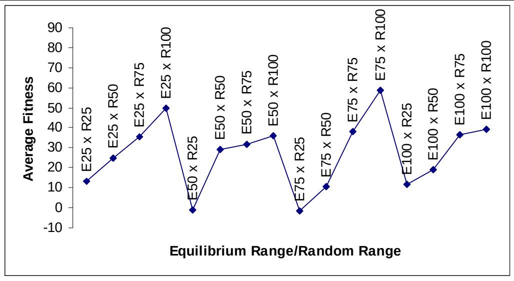


Mutation Size Versus Equilibrium Range

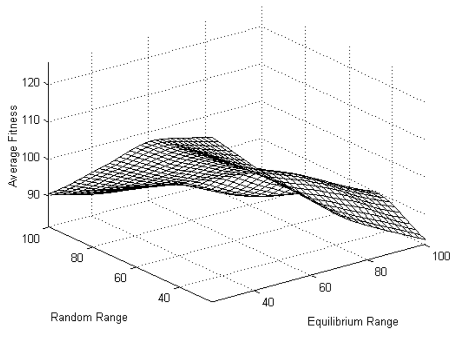
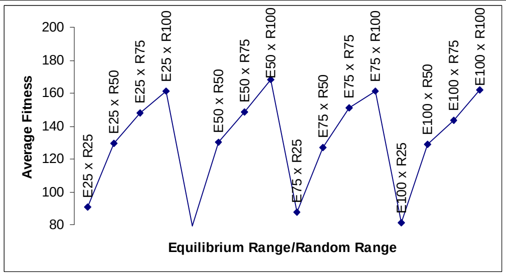
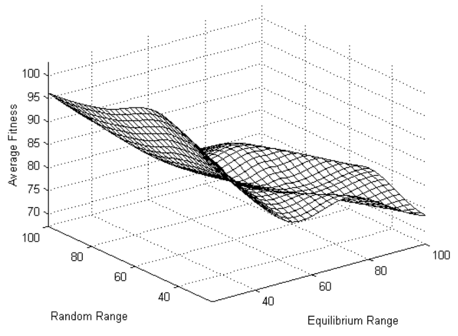
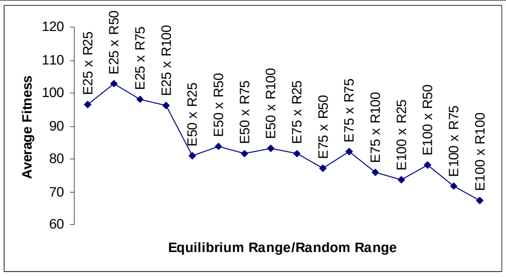
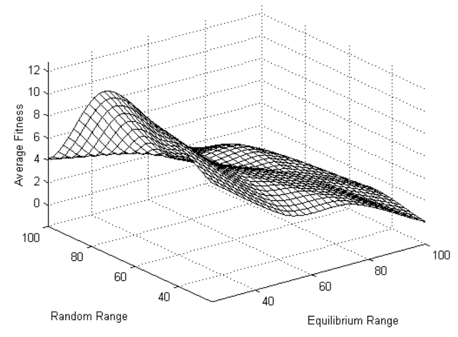
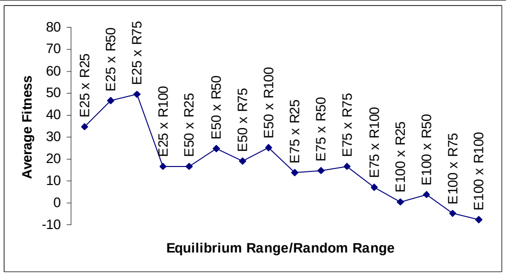
Normal Distribution with Simple Acceptance

	3D View	2D View
Mutate All		
Mutate One		
Mutate Some		

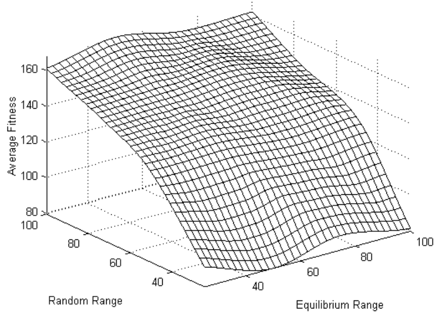
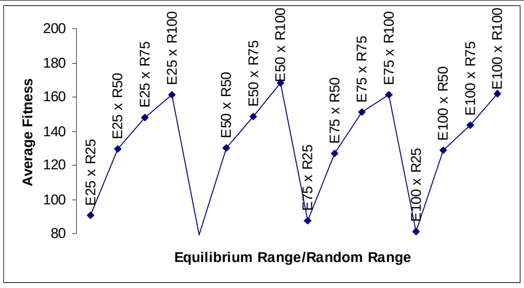
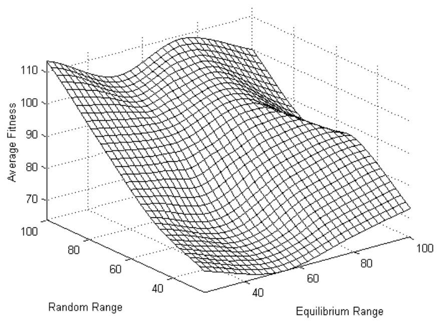
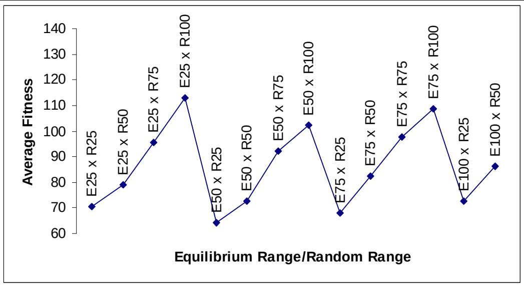
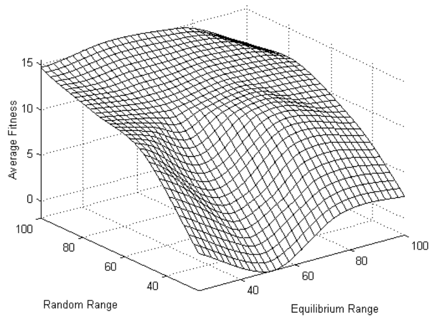
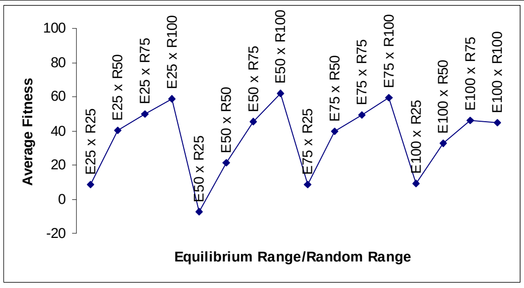
Normal Distribution with Simulated Annealing

	3D View	2D View
Mutate All		
Mutate One		
Mutate Some		

Uniform Distribution with Simple Acceptance

	3D View	2D View
Mutate All		
Mutate One		
Mutate Some		

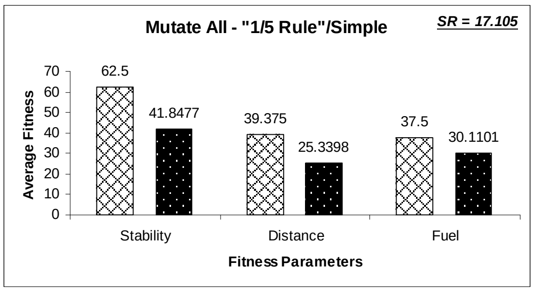
Uniform Distribution with Simulated Annealing

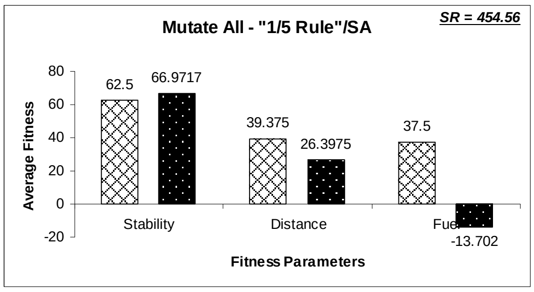
	3D View	2D View
Mutate All		
Mutate One		
Mutate Some		

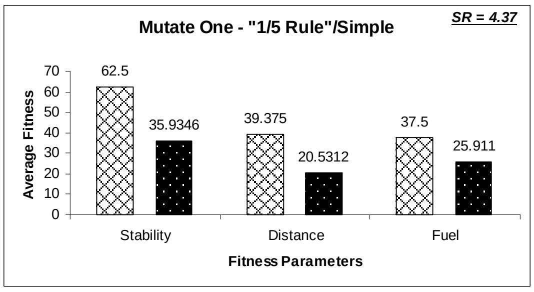
Implication of "1/5 Rule"

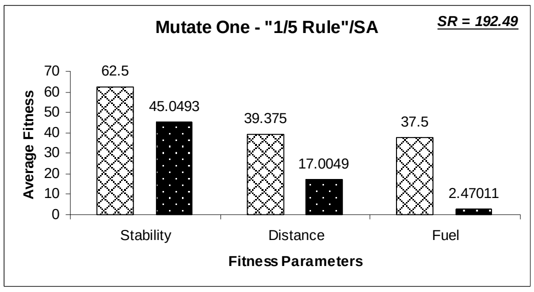
▨ Average Expected

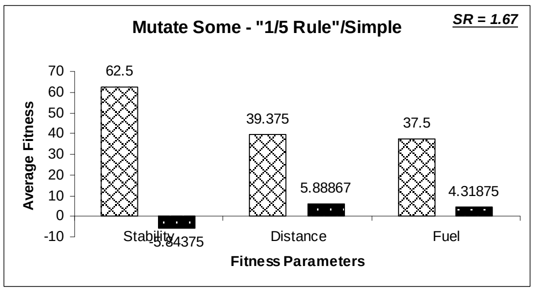
■ Average Evolved

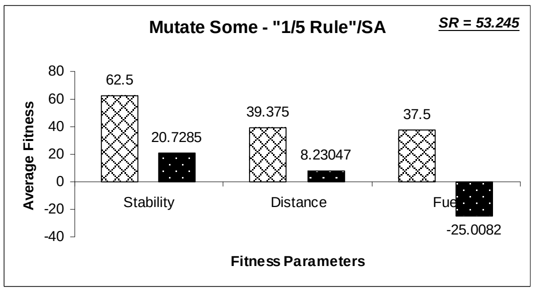












Appendix C

A Description of the Software used in this Project

During the course of the PhD, several programs were created to test the various systems. In this section, only the final program developed is shown. This program was created after testing the following modules (which were programmed using Borland Builder C++) individually and combining them together.

1. Neural Network
2. Robot
3. RTEAs

After testing that all these modules were working, an automated experimental tool (program) was developed for each RTEA. The features of the tool are listed below.

1. Conducts a range of experiments for each RTEA
    a. Mutate-All RTEA – 18562 experiments
    b. Mutate-One RTEA – 18562 experiments
    c. Mutate-Some RTEA – 4608 experiments

2. Stores the output information of each experiment in a file
    a. Expected Fitness and Evolved Fitness

$$\left.\begin{array}{rl} \text{i.} & \text{Normal/Simple/SA} \\ \text{ii.} & \text{Uniform/Simple/SA} \\ \text{iii.} & \text{“1/5 Rule”/Simple/SA} \end{array}\right\} \begin{array}{l} \text{1. Stability} \\ \text{2. Distance} \\ \text{3. Fuel} \end{array}$$

    b. Mutation size versus equilibrium range
        i. Normal/Simple/SA and Uniform/Simple/SA
            1. Average of Stability, Distance and Fuel

The experimental flowchart is shown in chapter 8 (figure 8.2). The pseudo code of the simulation method is shown overleaf.

```
1) Initialize Experimental Parameters;
2) Nested for Loop to generate the different experimental
combinations[1*];
      a) Run Experiment;
            i) Setup Network Parameters;
            ii) Setup Robot Parameters;
            for (Number of Simulation Cycles)
            {
              for (Each Number of Time Steps)
              {
                Simulate Network;
                Simulate Robot;
              }
              Calculate Fitness;
              Accept Changes According to the Required Condition;
              Mutate Network Parameters;
            }
      b) Store Results
```

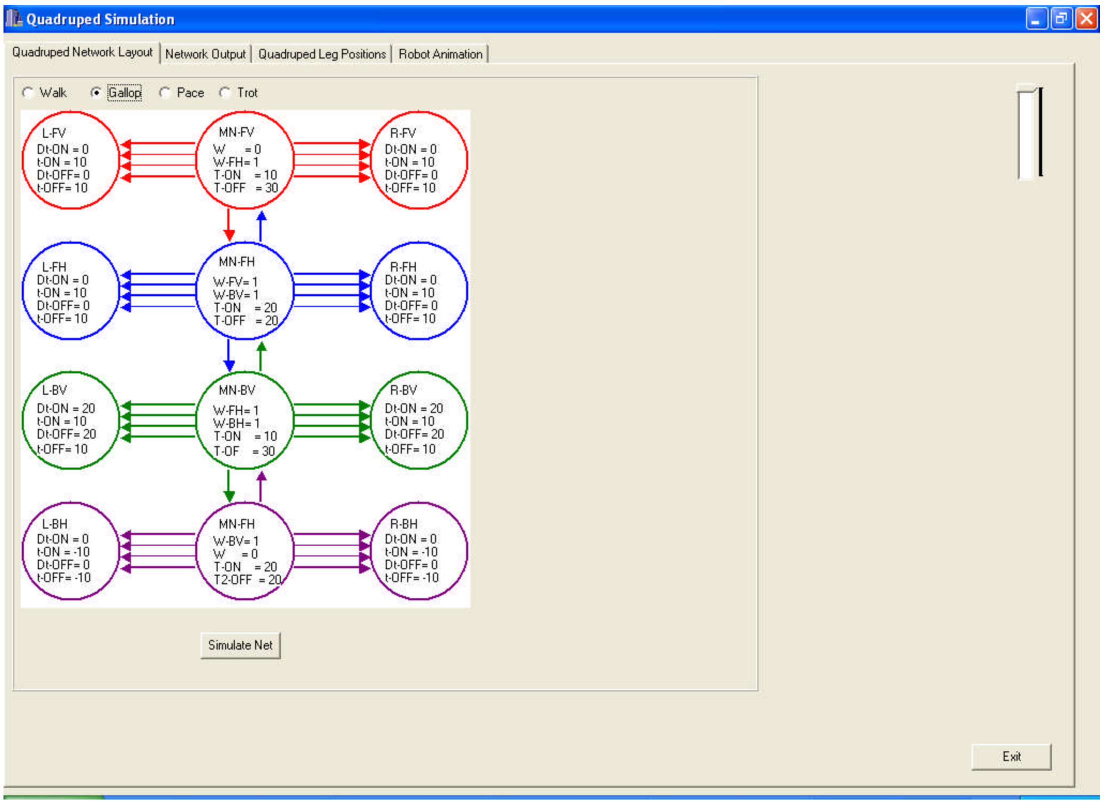
[1*]Nested *for* loop (Refer to Experimental Parameter Table in chapter 8, figure 8.1 – Page No. 120)

```
for (a = Each Random Number Type)
{
  for (b = Each Condition of Acceptance)
  {
    for (c = Each %of Equilibrium Range)
    {
      for (d = Each %of Mutation Size)
      {
        for (e = Each %of System Disturbance)
        {
          for (f = Each %of Required Stability)
          {
            for (g = Each %of Required Distance)
            {
              for (h = Each %of Required Stability)
              {
                Experimental Parameters(a,b,c,d,e,f,g,h);
              }
            }
          }
        }
      }
    }
  }
}
```

Appendix D

Screen Shots of Quadruped Simulation

Gallop Gait



Walk [✓ Gallop] Pace Trot

L-FV
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

MN-FV
W = 0
W-FH= 1
T-ON = 10
T-OFF = 30

R-FV
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

L-FH
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

MN-FH
W-FV= 1
W-BV= 1
T-ON = 20
T-OFF = 20

R-FH
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

L-BV
Dt-ON = 20
t-ON = 10
Dt-OFF= 20
t-OFF= 10

MN-BV
W-FH= 1
W-BH= 1
T-ON = 10
T-OF = 30

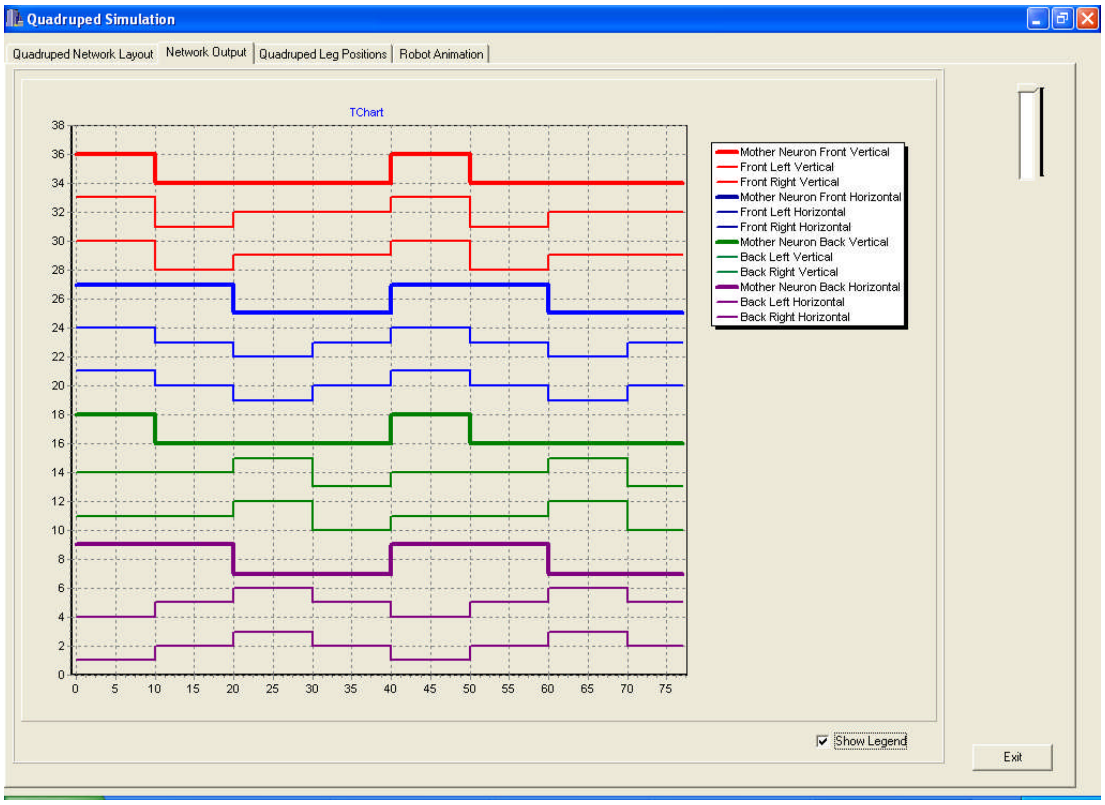
R-BV
Dt-ON = 20
t-ON = 10
Dt-OFF= 20
t-OFF= 10

L-BH
Dt-ON = 0
t-ON = -10
Dt-OFF= 0
t-OFF= -10

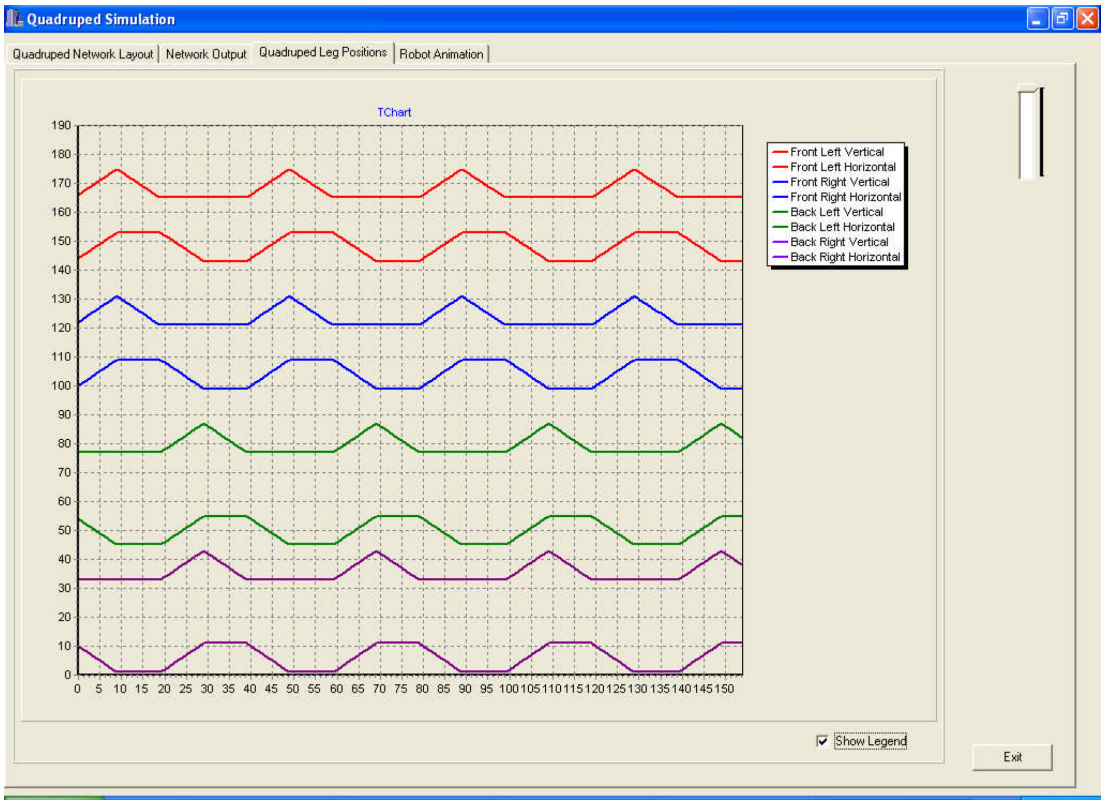
MN-FH
W-BV= 1
W = 0
T-ON = 20
T2-OFF = 20

R-BH
Dt-ON = 0
t-ON = -10
Dt-OFF= 0
t-OFF= -10

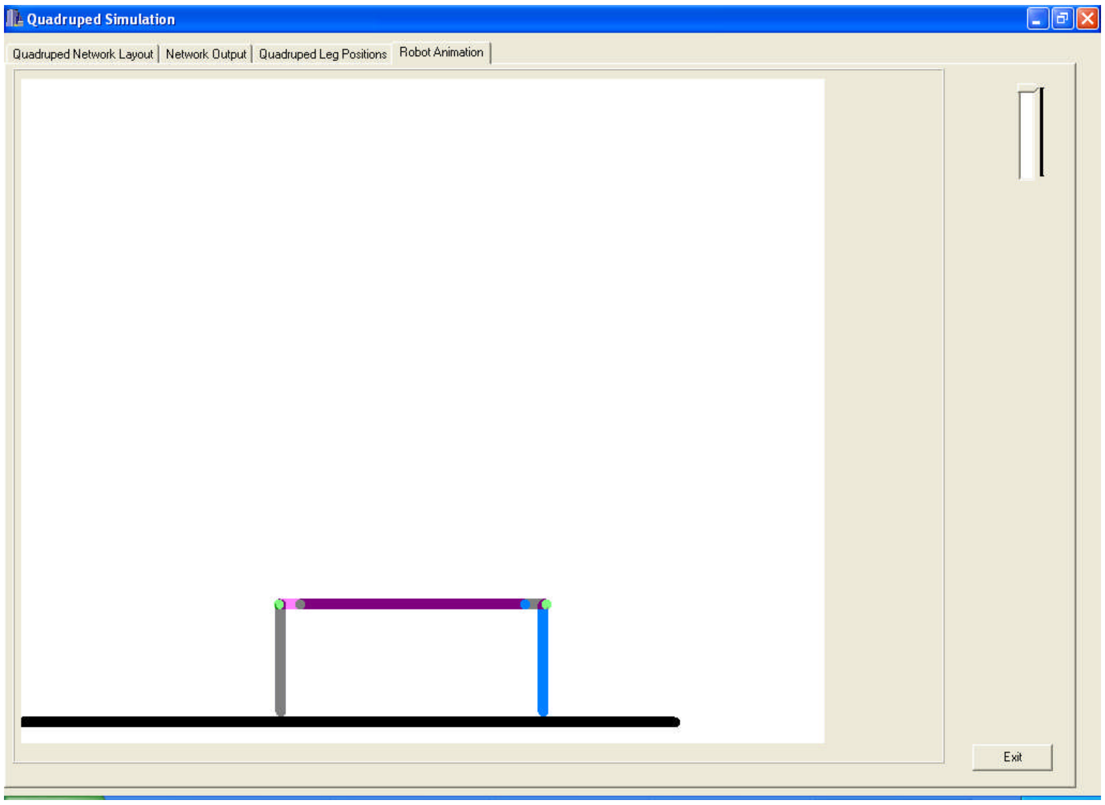
Gallop Gait Network Parameter



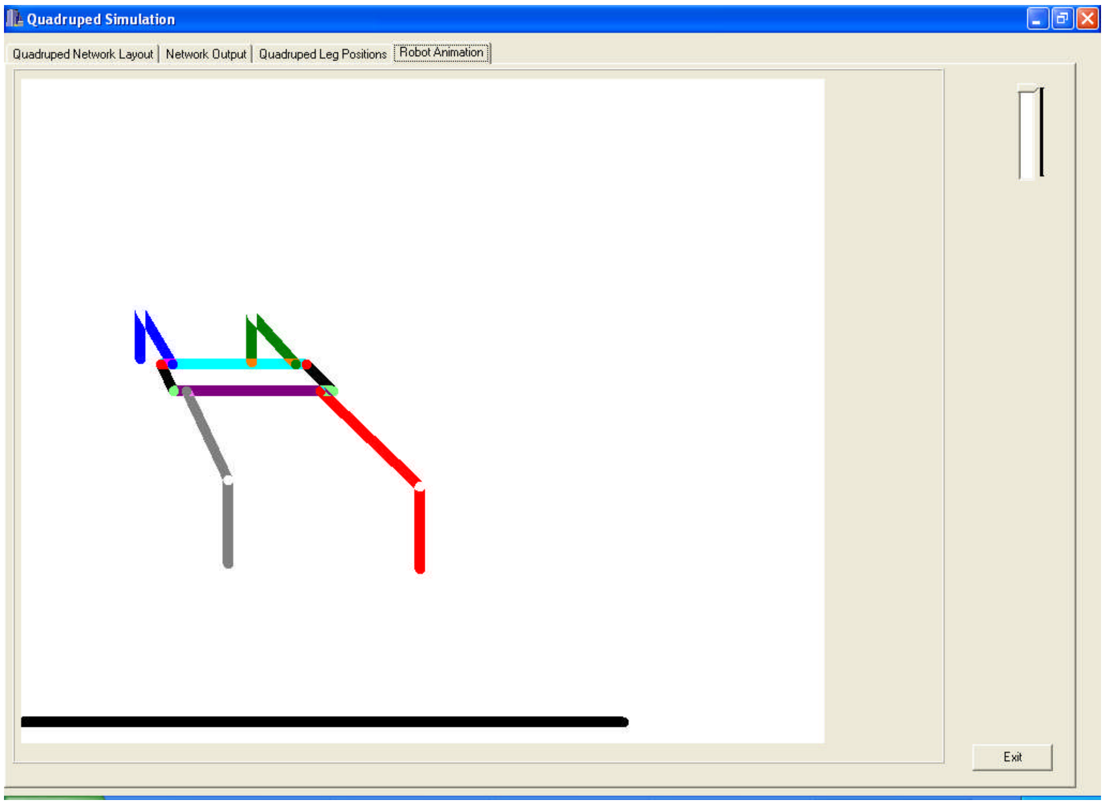
Network Output



Quadruped Leg Positions

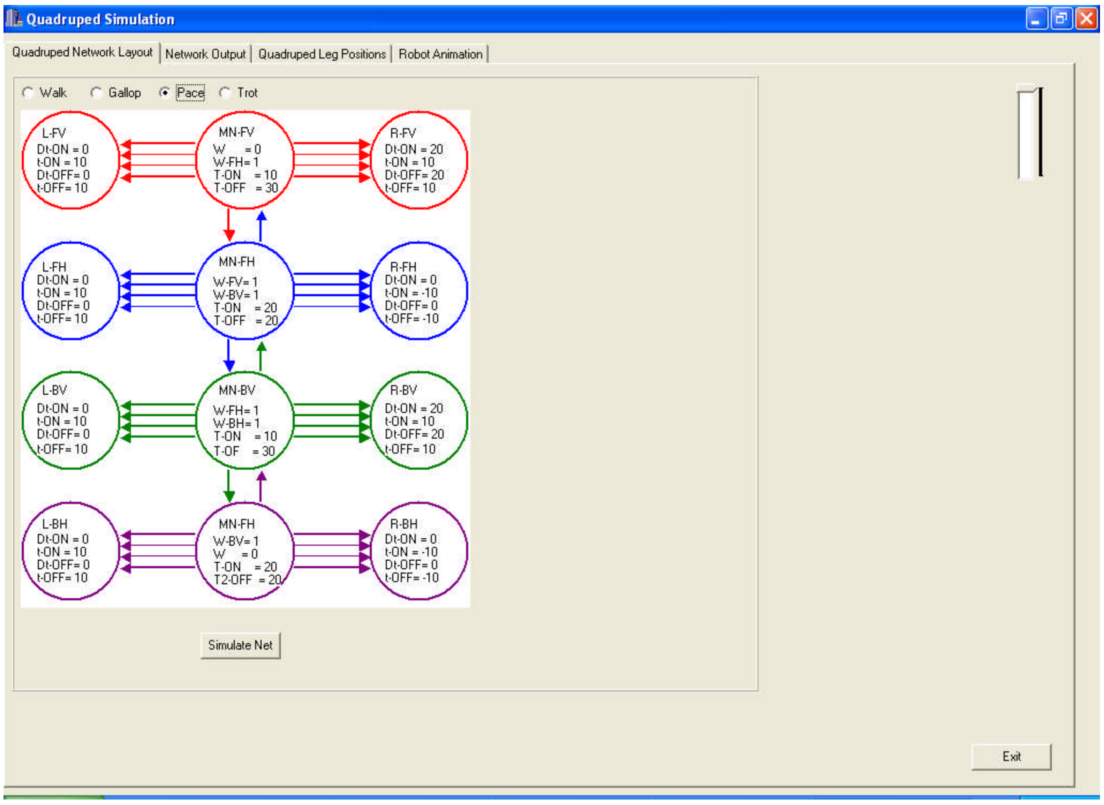


Side View of Quadruped Robot



3D View of Quadruped Robot

Pace Gait


Quadruped Simulation

Quadruped Network Layout | Network Output | Quadruped Leg Positions | Robot Animation

Walk Gallop [✓ Pace] Trot

L-FV
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

MN-FV
W = 0
W-FH= 1
T-ON = 10
T-OFF = 30

R-FV
Dt-ON = 20
t-ON = 10
Dt-OFF= 20
t-OFF= 10

L-FH
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

MN-FH
W-FV= 1
W-BV= 1
T-ON = 20
T-OFF = 20

R-FH
Dt-ON = 0
t-ON = -10
Dt-OFF= 0
t-OFF= -10

L-BV
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

MN-BV
W-FH= 1
W-BH= 1
T-ON = 10
T-OF = 30

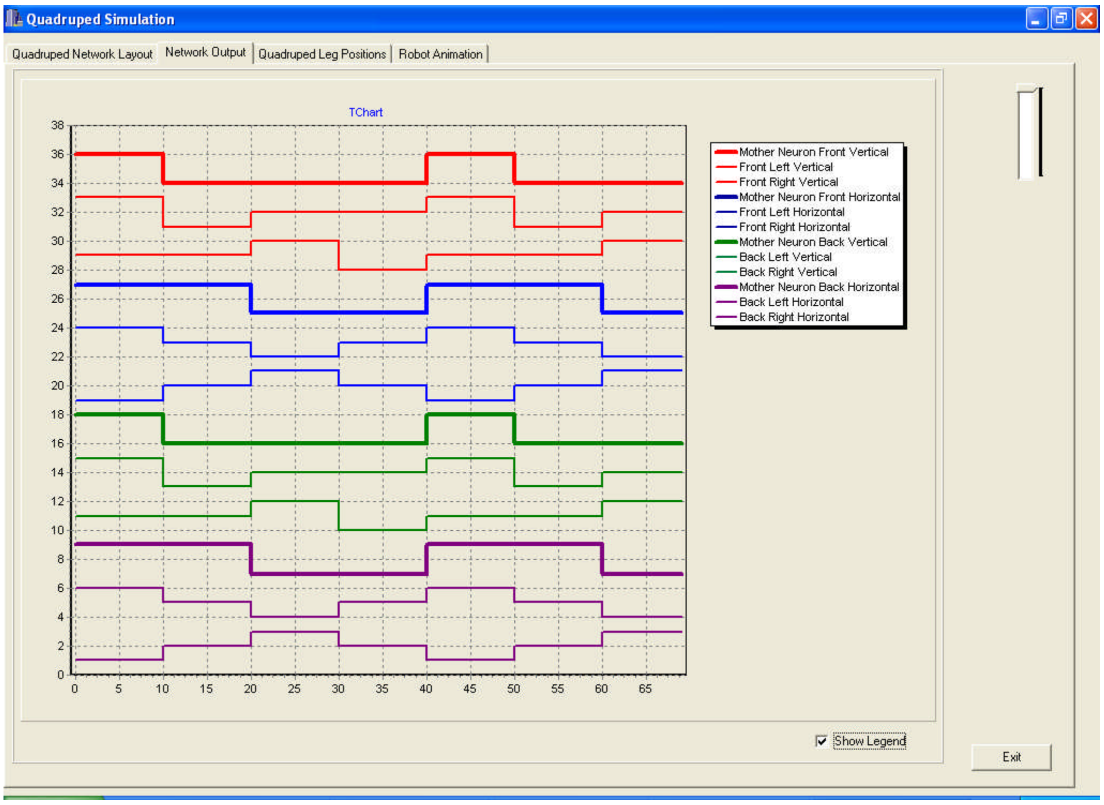
R-BV
Dt-ON = 20
t-ON = 10
Dt-OFF= 20
t-OFF= 10

L-BH
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

MN-FH
W-BV= 1
W = 0
T-ON = 20
T2-OFF = 20

R-BH
Dt-ON = 0
t-ON = -10
Dt-OFF= 0
t-OFF= -10

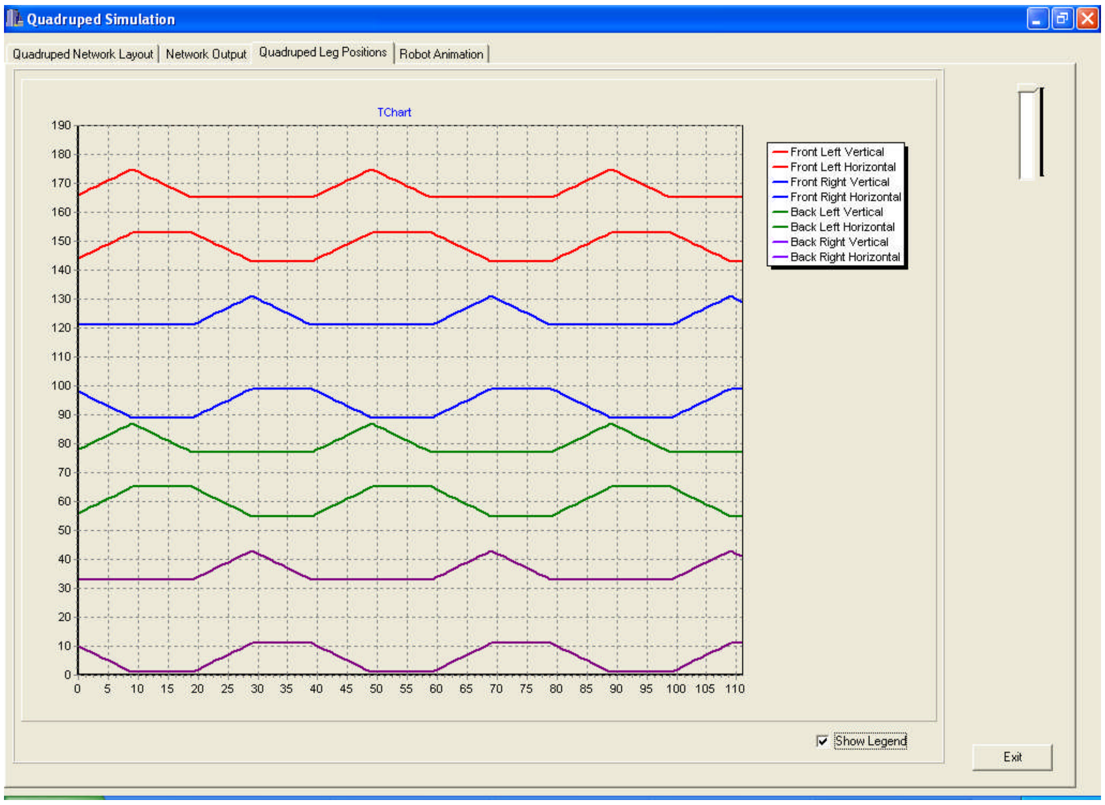
Pace Gait Network Parameters


Quadruped Simulation

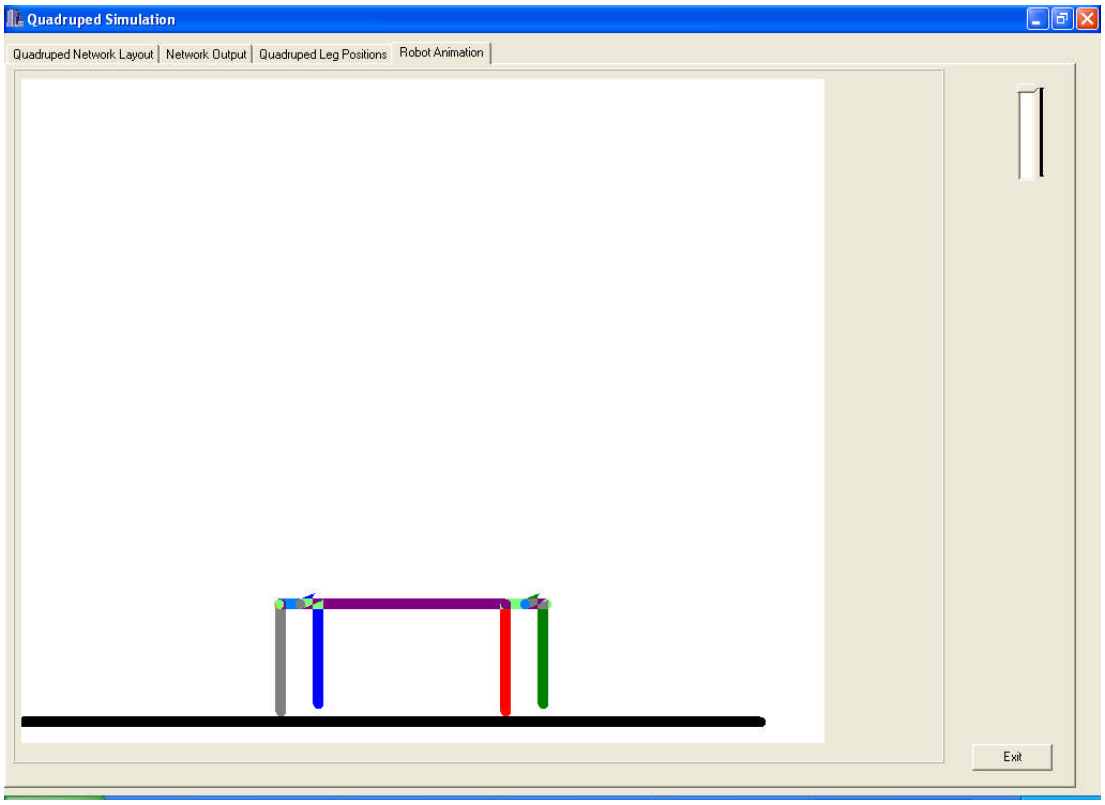
Quadruped Network Layout | Network Output | Quadruped Leg Positions | Robot Animation

[✓ Show Legend]

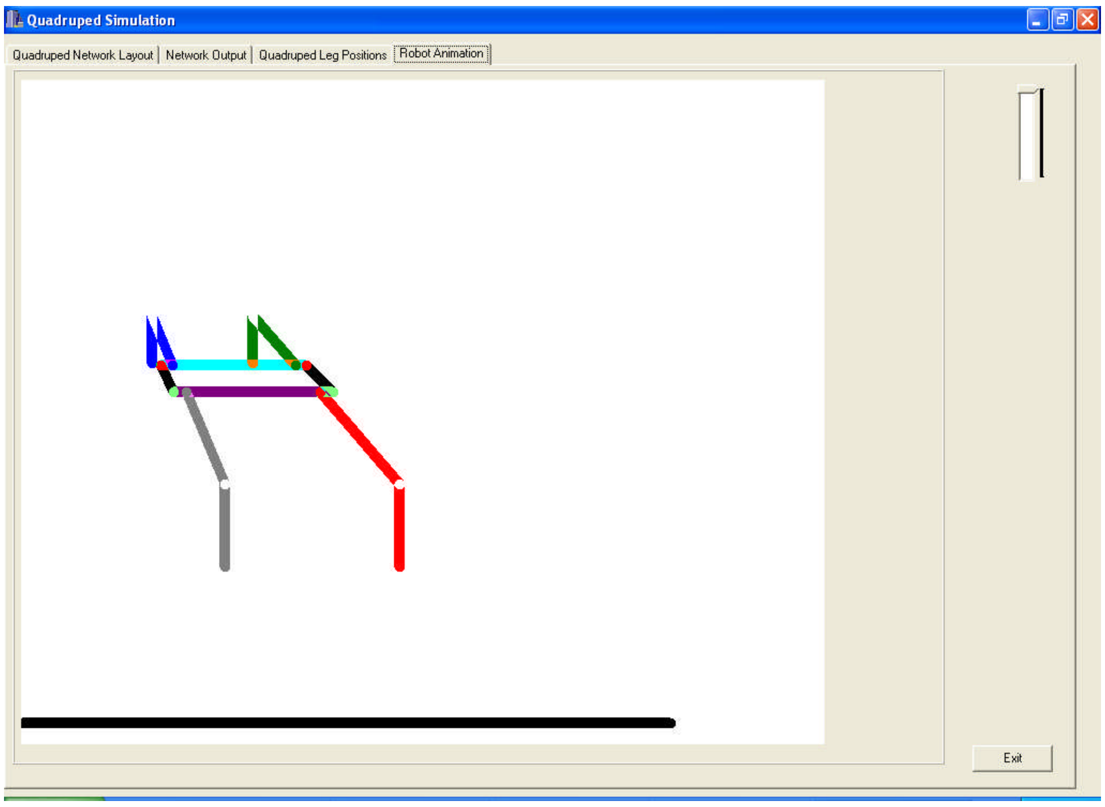
Network Output



Quadruped Leg Positions

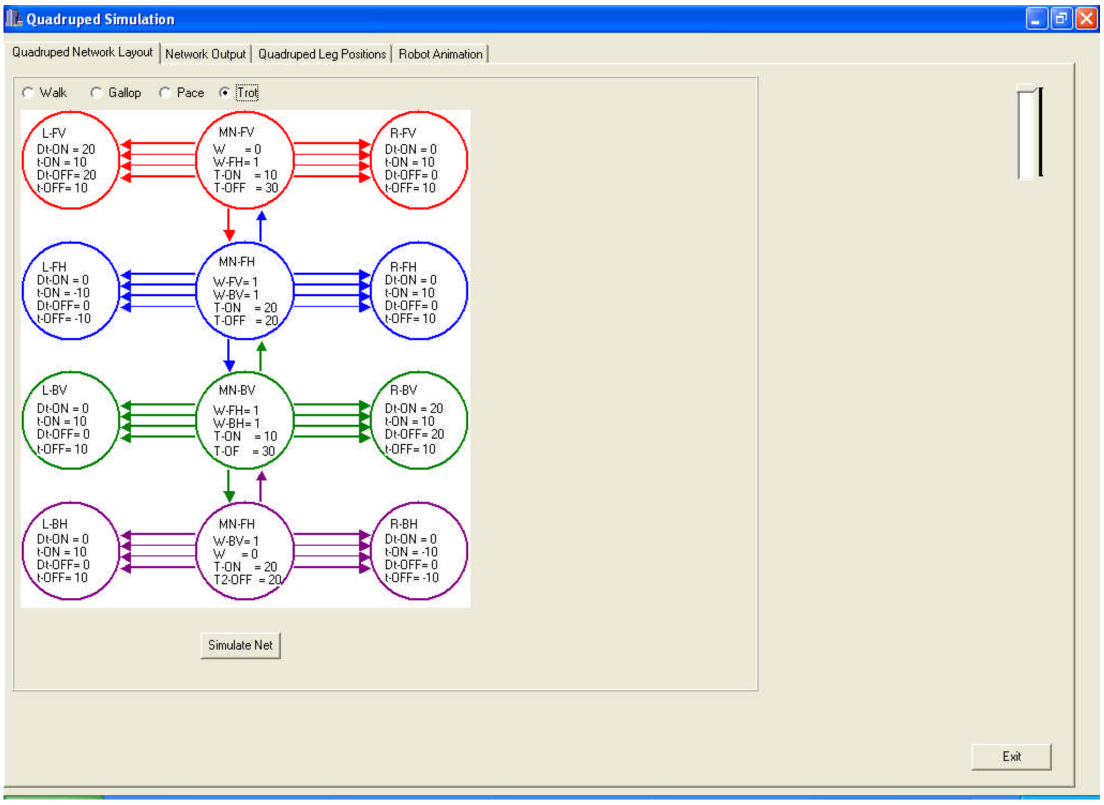


Side View of Quadruped Robot



3D View of Quadruped Robot

Trot Gait



Walk Gallop Pace [✓ Trot]

L-FV
Dt-ON = 20
t-ON = 10
Dt-OFF= 20
t-OFF= 10

MN-FV
W = 0
W-FH= 1
T-ON = 10
T-OFF = 30

R-FV
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

L-FH
Dt-ON = 0
t-ON = -10
Dt-OFF= 0
t-OFF= -10

MN-FH
W-FV= 1
W-BV= 1
T-ON = 20
T-OFF = 20

R-FH
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

L-BV
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

MN-BV
W-FH= 1
W-BH= 1
T-ON = 10
T-OF = 30

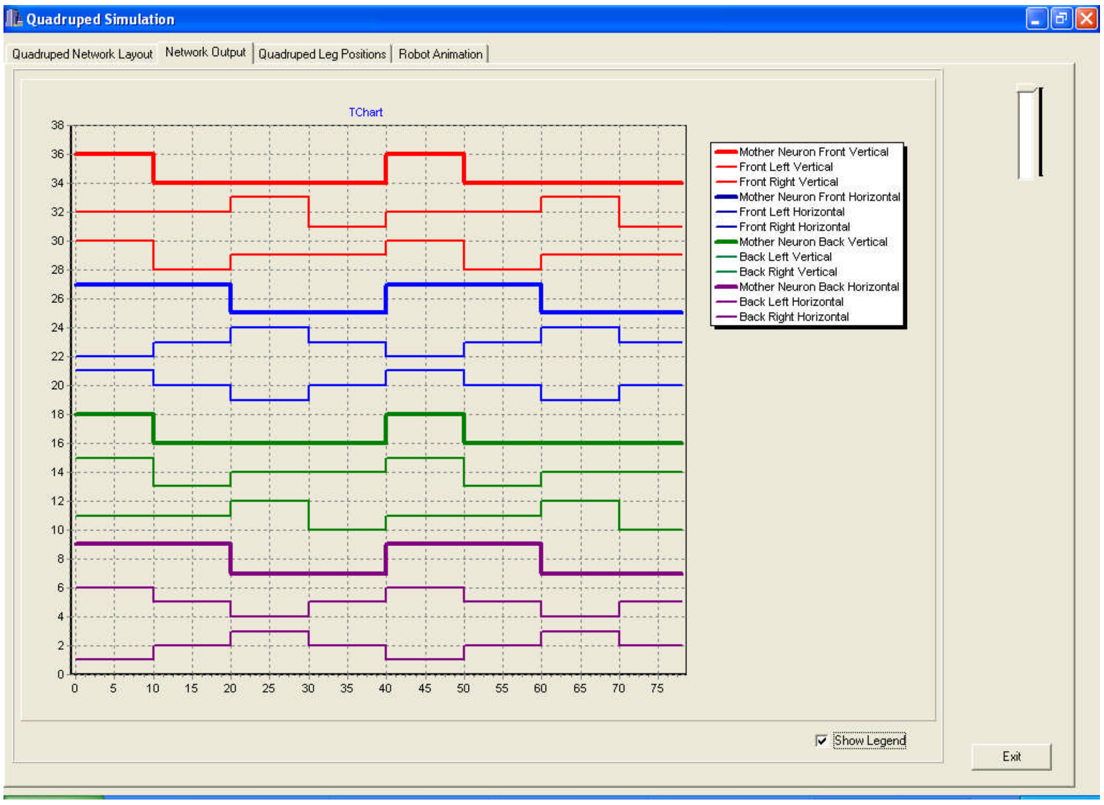
R-BV
Dt-ON = 20
t-ON = 10
Dt-OFF= 20
t-OFF= 10

L-BH
Dt-ON = 0
t-ON = 10
Dt-OFF= 0
t-OFF= 10

MN-FH
W-BV= 1
W = 0
T-ON = 20
T2-OFF = 20

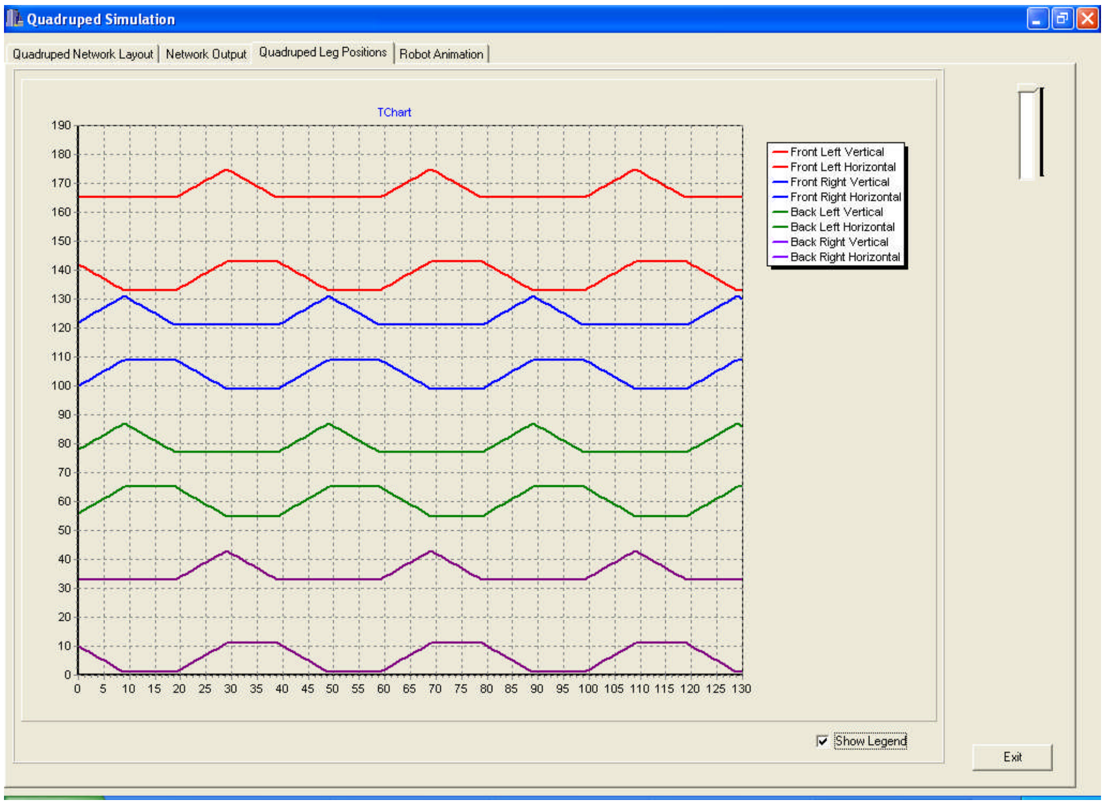
R-BH
Dt-ON = 0
t-ON = -10
Dt-OFF= 0
t-OFF= -10

Trot Gait Network Parameters

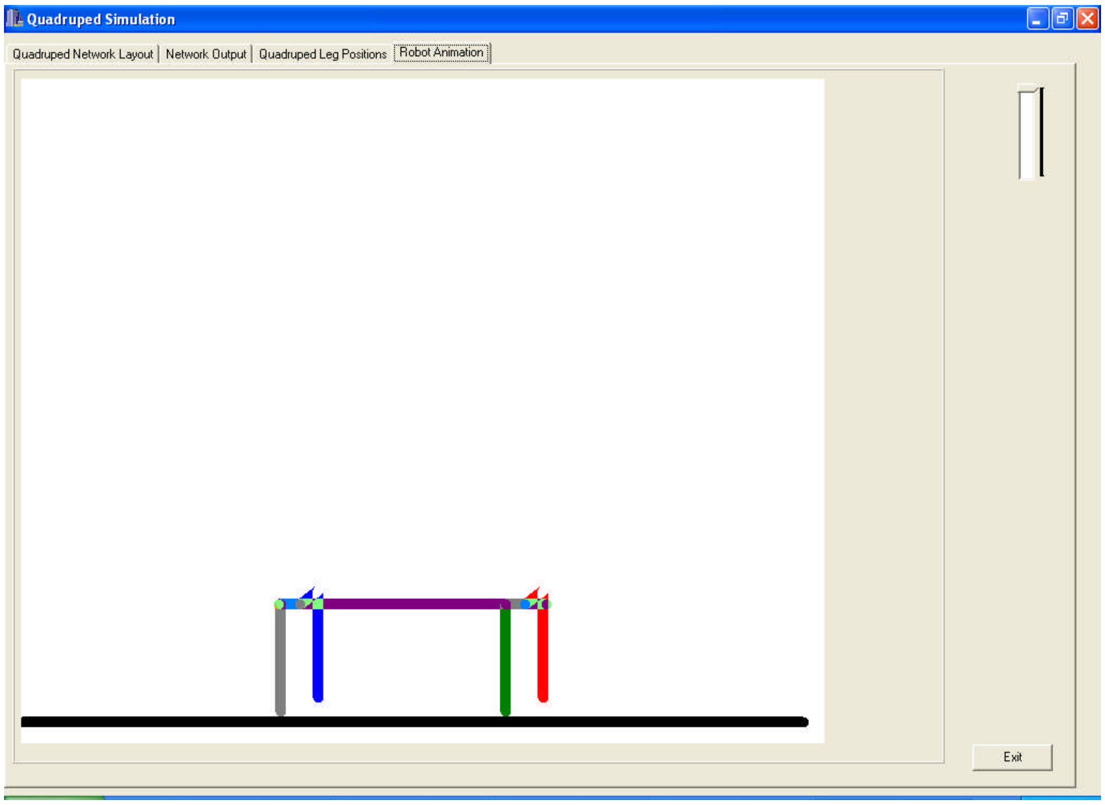


[✓ Show Legend]

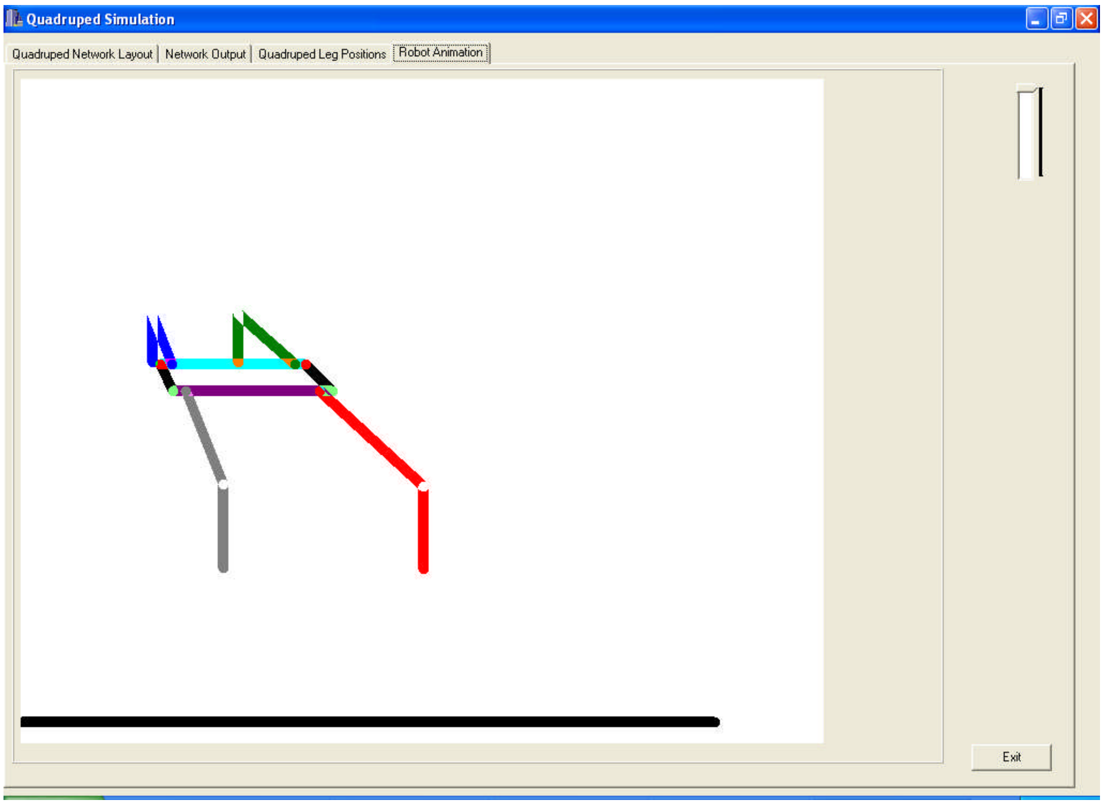
Network Output



Quadruped Leg Positions



2D View of Quadruped Robot



3D View of Quadruped Robot