An Experimental Design Framework for Evolutionary Robotics

Mr. Robert McCartney B.Eng.(Hons.)

Submitted for assessment for the award of the degree of
Masters in Electronic Engineering by Research and Thesis.

Supervisor: Dr. Barry McMullin.

School of Electronic Engineering.

Dublin City University.

June 1996.

Volume 1 of 1

To my family...

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Masters in Engineering (M.Eng.) is entirely my own work and has not been taken from the works of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed : ________________________________

ID No.: 93700261

Date : ____________

Abstract

An Experimental Design Framework for Evolutionary Robotics

Robert McCartney B.Eng. (Hons.)

Based on the failures of work in the area of machine intelligence in the past, a new paradigm has been proposed: for a machine to develop intelligence it should be able to interact with and survive within a hostile dynamic environment. It should therefore be able to display adaptive behaviour and respond correctly to changes in its situation. This means that before higher cognitive properties can be modeled, the modeling of the lower levels of intelligence would be achieved first. Only by building on this platform of physical and mental abilities may it be possible to develop true intelligence. One train of thought for implementing this is to control and design a robot by modeling the neuroethology of simpler animals such as insects.

This thesis outlines one approach to the design and development of such a robot, controlled by a neural network, by combining the work of a number of researchers in the areas of machine intelligence and artificial life. It involves Rodney Brooks' subsumption architecture, Randall D. Beer's work in the area of computational neuroethology, Richard Dawkins' work in the area of biomorphs and computational embryology and finally the work of John Holland and David Goldberg in genetic algorithms.

This thesis will demonstrate the method and reasoning behind the combination of the work of the above named researchers. It will also detail and analyse the results obtained by their application.

List of Figures

Preface

This thesis represents the combination and completion of a number of works. It was completed with the School of Electronic Engineering while registered as a student of the Integrated B.Eng./M.Eng. study programme. For that reason a number of other documents relating to this project have previously been produced [23,24,25,26]. The integrated programme allows students to initiate research for the award of a Masters degree in Engineering while registered as an undergraduate.

1. Introduction

This thesis is about machine intelligence. It has been inspired by the lack of success in recent years in the areas of connectionism, neural networks and expert systems. All of these areas have promised much but unfortunately delivered very little. None of these areas have made significant progress in developing systems which display an intelligence that is not either defined within strict operational boundaries or uses simplistic, representationalised input data. Recently however, a number of researchers have attempted to approach the problem of modeling machine intelligence from a new direction. The new direction which they propose is very simple and is the foundation stone upon which this thesis is based. They propose that, in order for a machine to exhibit higher level cognitive properties, it is first essential that the machine be able to deal with the real environment in which it exists.

The evolution of human intelligence is worth considering at this point as this is the intelligence that is referred to when people discuss the creation of artificial intelligence. The planet Earth is approximately 4.6 billion years old, and single cell life first appeared on it about 3.5 billion years ago The first photosynthetic plants appeared about 1 billion years later. Two billion years after that, the first vertebrate animals and fish appeared and then about 450 million years ago insects appeared. Reptiles were around about 370 million years ago and mammals arrived only about 250 million years ago. The human race appeared approximately 2.5 million years ago; descended from the first apes who appeared only 16 million years before that. Human level intelligence only first became apparent with the discovery of agriculture some 19,000 years ago, writing about 5,000 years ago and "expert" knowledge only in the last few centuries. This means that evolution has spent only 0.005% of 3.5 billion years of the evolutionary time span dealing with higher level intelligence. This could be seen to suggest that the capabilities of problem solving, language, reason and expert knowledge are either made more simple as a result of, or alternatively dependant on, the ability of a being to deal interactively with the hostile dynamic environment in which it finds itself.

In continuation with this theme, a machine intelligence should not be designed within a cotton wool model of the real world and then be expected to be capable of dealing with the real world at some later stage. The only valid model for the real world in all its chaotic glory is the real world itself. The functional and reality gap between simulation of intelligent behaviour and the implementation of intelligent behaviour is too great. In everyday operation an intelligent machine should be able to adapt to changes in its situation and environment. It is important that it be able to protect itself from physical damage (for example stop itself driving off a cliff if a bridge that existed the day before was now no longer present). The machine must be able to differentiate between, and deal with, different classes of problems and come to an optimum, scenario dependant solution, rather than simply follow an algorithmic path to a pre-defined answer in a pre-defined situation.

Therefore, it is essential that the machine be multi-tasking and fully aware of the world. For example, even taking a parcel from a position x to another position y requires a vast amount of physical and mental capabilities. The controlling intelligence must be continually prioritising problems and coming to optimum solutions to complete the task. To illustrate, consider some of the tasks which must be completed. Detect and grasp the parcel, plan route from x to y based on internal (or external) area map, monitor terrain to avoid becoming stuck or damaged, continually choose optimum path around obstacles, monitor progress, monitor position and physical condition, etc. all while remembering its primary goal of delivering the parcel.

The next question is, of course, how do we design a robot that can behave like this? Due to the incredible complexity of the human brain it is currently impossible to model it properly. So a number of researchers have proposed that to design or to create a machine intelligence comparable with human intelligence it is first essential to model the simpler intelligence of simple animals such as insects. Using this modeling and neurobiological knowledge it should then be theoretically simpler to work upwards from there. This thesis defines and evaluates a single approach to the development of a framework for the development of a low level adaptive machine intelligence which could be continually updated and augmented. The framework is designed to create a machine that may be continually improved using new found knowledge in the areas of biology, robotics, artificial intelligence and natural systems modeling.

This thesis is divided into a number of different sections. These are:

1. Thesis Overview.
2. Background theory and description of:
    - Subsumption Architecture.

- Genetic Algorithms.
- Computational Embryology.
- Computational Neuroethology.

3. Details of:
    - Adaptation of background theory to application.
    - Hardware descriptions of robot, controlling microprocessor board and interface.
    - Software written for application.

4. Results and analysis.
    - Description.
    - Evaluation.
    - Recommendations

5. Conclusion.

2. Background Theory

2.1 Introduction

In this section, the theories, ideas and inspirations behind the project will be outlined. More detailed knowledge of these topics can be obtained from the references given, as it is not feasible to provide more precise details and/or examples within the confines of this thesis.

The work of Rodney Brooks[3,4,5] whose subsumption architecture idea is one of the cornerstones of the masters degree project, is outlined in section 2.2. The work of Randall D. Beer[2] who has designed and simulated the operation of neural networks based on a computational neuroethological approach, along with the modifications that were necessary for its application to this project are detailed in section 2.3. Finally, based on the work of Goldberg[14] and Dawkins[9,10,11], genetic algorithms and their application to this project is presented in sections 2.4 and 2.5.

The information in these sections is not sufficient to understand the complexities of the fields; rather it serves only as a brief introduction to their basic concepts. This is to make the theoretical basis of the project more tangible: to bring together, and show the relationship between, all the different aspects of the project work being done

2.2 Subsumption Architecture

2.2.1 Introduction

As stated above, one of the main sources of inspiration for the Master's project and which underlies the work done is the work of Rodney Brooks[3,4,5]. Brooks' idea is that existing approaches to the development of machine intelligence are fundamentally flawed. He proposes that in order for a machine to develop intelligence it is first essential that the machine be able to deal successfully with a hostile external environment. To this end he has proposed a new design structure for intelligent machines which is based on a behavioural design methodology rather than the more accepted functional design decomposition currently used by many researchers in the areas of artificial intelligence and Robotics. In this section the two approaches to the design of controlling frameworks will be compared and the advantages of the subsumption architecture approach described.

2.2.2 Functional Decomposition

Figure 2.1 represents a functional decomposed design structure for generating 'intelligent' behaviour in machines. The robot control algorithm designed using this approach would incorporate as many solutions as were necessary/possible to enable the robot to interact with its environment and thus exhibit some form of intelligent behaviour. However, due to the serial nature of this structure as well as the complex interactions and message passing techniques employed by this form of design, it fails. This failure is illustrated by the fact that if any particular section of the robot were to fail or become so obsolete as to be rendered useless then an entirely new robot would have to be designed and built to overcome the failure or to upgrade the hardware. This is at the expense of money, materials and time. Obviously from a robustness, as well as a practical viewpoint, this method of design is unacceptable in the long run.

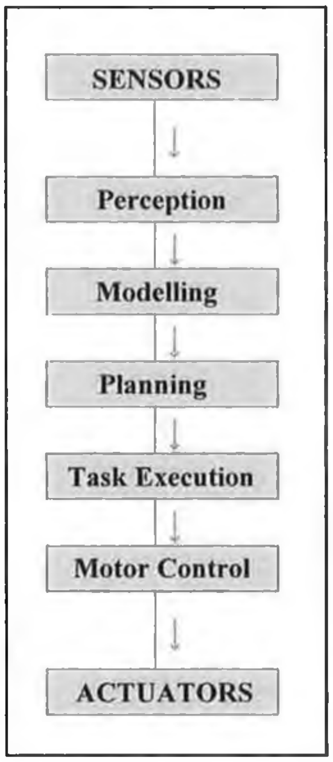


Figure 2.1 Functional Decomposition Design Structure

2.2.3 Behavioural Decomposition

However, shown in figure 2.2 is a diagram which describes a much different approach to designing a robot which demonstrates at least the same functionality as the functionally designed one. From the diagram it can be seen that rather than a single link connecting the sensory input to the output, there are a number of parallel links. Each of these links is graded on a behavioural level and each level uses the functionality of the previous levels to carry out its own tasks. To illustrate; the construction of such a robot begins with the design of a very simple robot which successfully implements its own low-level behavioural tasks such as the avoidance of objects. When this level of behaviour has been successfully implemented, tested and proven within a real environment, the next level of behaviour is designed. This level could be, for example, a wandering behaviour. To enable the robot to apparently wander around its environment a level of control is added which takes advantage of the lower level's behavioural capabilities. It subsumes control of the lower level. This robot is then tested fully and when it has been found to wander successfully, the next level of behavioural control (perhaps an environmental mapping behaviour) is designed, added and tested. The addition of each of the completed stages offers a higher level of overall complexity and intelligent behaviour to the robot.

Each level of the architecture operates independently of the others but each level can subsume control of the levels lower than itself in the behavioural hierarchy and use them to its own advantage. This process, Brooks believes, will eventually lead to a machine which can make its own decisions on abstract, logical and pure reflex levels and thus potentially demonstrate an intelligence far outstripping anything currently implemented by functionally designed robots.

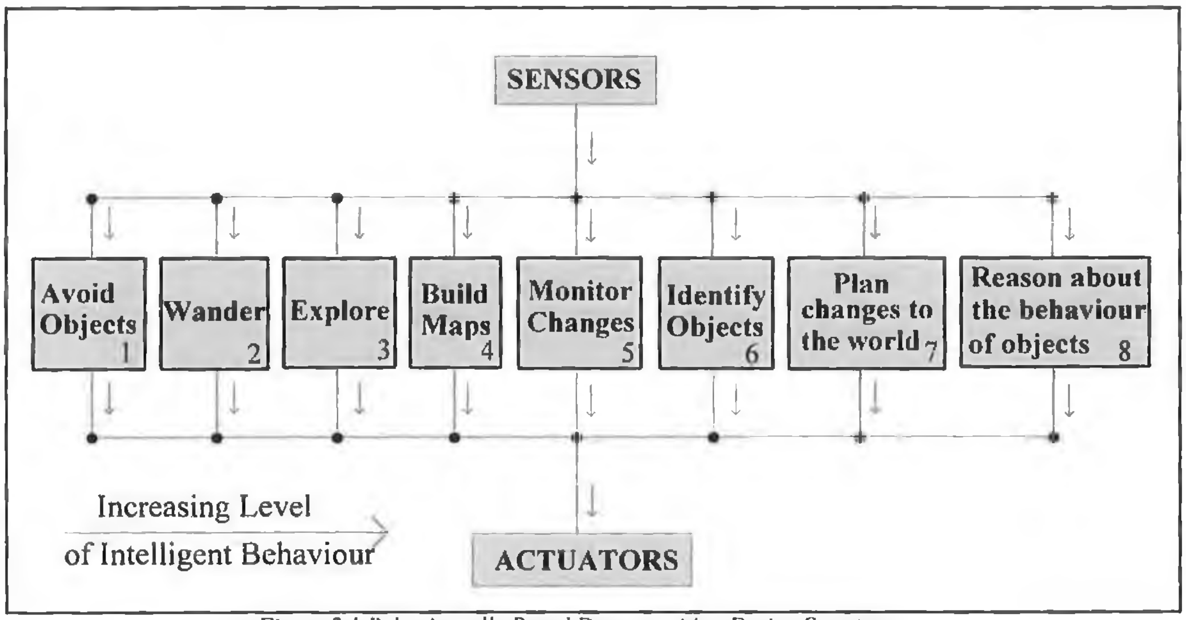

SENSORS
Avoid Objects 1
Wander 2
Explore 3
Build Maps 4
Monitor Changes 5
Identify Objects 6
Plan changes to the world 7
Reason about the behaviour of objects 8
Increasing Level of Intelligent Behaviour
ACTUATORS

Figure 2.1 Behaviourally Based Decomposition Design Structure.

2.2.4 Advantages/Disadvantages of Subsumption Architecture

The beauty of the subsumption architecture is that, due to the modular nature of both the robots intelligence and construction, should any particular level be found to be faulty or technologically redundant, then only the offending section need be redesigned - not the entire robot. This saves on money, materials and time. One substantial drawback however is that a great deal of parallel computational power is potentially necessary to implement such a structure. As a result, the robot could (in today's world) be quite expensive to implement initially. However, due to the ease of maintenance and upgrade involved with a truly subsumptive robot design, the architecture prevents (as much as possible) the robot becoming obsolete due to the failure or redundancy of a single section.

2.2.5 Application of Subsumption Architecture to the Project

Taken in the form as described by Brook, subsumption architecture is a long term design configuration. Brooks used Finite State Machines to implement each behavioural level in the architecture. However, as is discussed in more detail in the next section, this is obviously not a very biologically inspired approach to the implementation of the subsumption architecture ideal. Subsumption architecture is used simply as a ‘container’ for all the other facets of the project. It is the primary cornerstone of the project but in itself is not implemented fully. To be implemented fully, a second level of behaviour would have to be successfully implemented above a successful first level. Within the context of the project only the first behavioural level was implemented and explored. Therefore, any further reference to subsumption architecture must be seen in that context.

2.3 Computational Neuroethology

Brooks' subsumption architecture control structure is based on the interaction of many Finite State Machines(FSMs)[4]. This offers an ease of implementation because the design of FSMs is not excessively complex if the problem is well defined. However (as was discussed in the introduction), in the context of this project it was decided to use a more biologically inspired choice of control structure for each level of behaviour. Following research in the area of neural networks, it was decided that they could offer what was required. The type of neural network control structure chosen to implement was a heterogeneous neural network structure. The design and construction of this form of neural network was first encountered in the work of Randall Beer[2]. In this work he describes the use of a technique known as computational neuroethology.

2.3.1 What is Computational Neuroethology?

Beer describes computational neuroethology as:

> *"...The direct use of behavioural and neurobiological ideas from simpler natural animals to construct artificial nervous systems for controlling the behaviour of autonomous agents"*
>
> [2][p xvi].

D. T. Cliff in his paper[8] also delivers a concise and studied discussion on computational neuroethology. His conclusion references the ability of networks designed using computational neuroethology to span the MacGregor-Lewis stratification[1] [26]. He references material common to the area of this thesis. Principally, he references extensively the work of Rodney Brooks. In this paper he provisionally defines computational neuroethology as the

> *"...study of neuroethology using the techniques of computational neuroscience".*
>
> [8]

In particular he notes that a very specific aspect of computational neuroethology is the

> *"...increased attention to the environment that the neural entity is a component of."*
>
> [8]

2.3.2 Beer's Work

In the previous section 2.2 on subsumption architecture, reference was made to Brooks' belief that work in the area of artificial intelligence was fundamentally flawed in its approach[3,4,5]. Beer, in his work, makes a very similar statement in the preface of his book[2], saying that thinking in this area

> *"...has been dominated by the notion that intelligence consists of the proper manipulation of symbolic representations of the world."*
>
> [2]

---

[1] A simple taxonomy of levels of analysis. Cliff however, makes reference to the stratification as being potentially non-ideal and perhaps requiring a further detailing of levels[8].

Certainly this seems to tie in very well on a conceptual level with Brooks' thesis of using the world as its own model[4]. It was viewed at the beginning of this project that a marriage of the work's of these two men would be both interesting as well as being, potentially, a rewarding approach. This reward being based on the combination of the real time efforts of Brooks[3,4,5] and the perceived mental processing offered by Beer's neural network structure[2]. Beer himself states that simpler animals possess a degree of adaptive behaviour that far exceeds that available to the most complex of artificial ones [2]. This level of processing power seemed ideal for a real time robotic implementation.

Beer successfully implemented and documented [2] a simulated insect. Its behaviour was due to a neural network which was based on a map of the understood neural mechanics of the insect *Periplaneta Americana*: the American cockroach. The simulated insect, using a neural network constructed using neural node models of the type described in figure 2.3 below, successfully traversed its simulated environment. It achieved some of its specified goals, such as food-finding, and it emulated the behaviour of many real insects by, for example, performing an edge following behavioural pattern around its environment.

Beer attributes this success to, among other things, the closeness of the neural model he employed to the structure of real neurons[2]. He refers to the work of Llinas[22] and emphasises the findings of Selverston[32]. Specifically he emphasises the fact that nerve cells contain a wide variety of active conductances which appear to allow them to demonstrate complex time-dependant behavioural responses to stimulation. They can also allow demonstrate apparently spontaneous activity when the network is active.

Selverston studied the interactions of active conductances at a cellular level and found that they appeared to be crucial to the function of neural circuits[32].

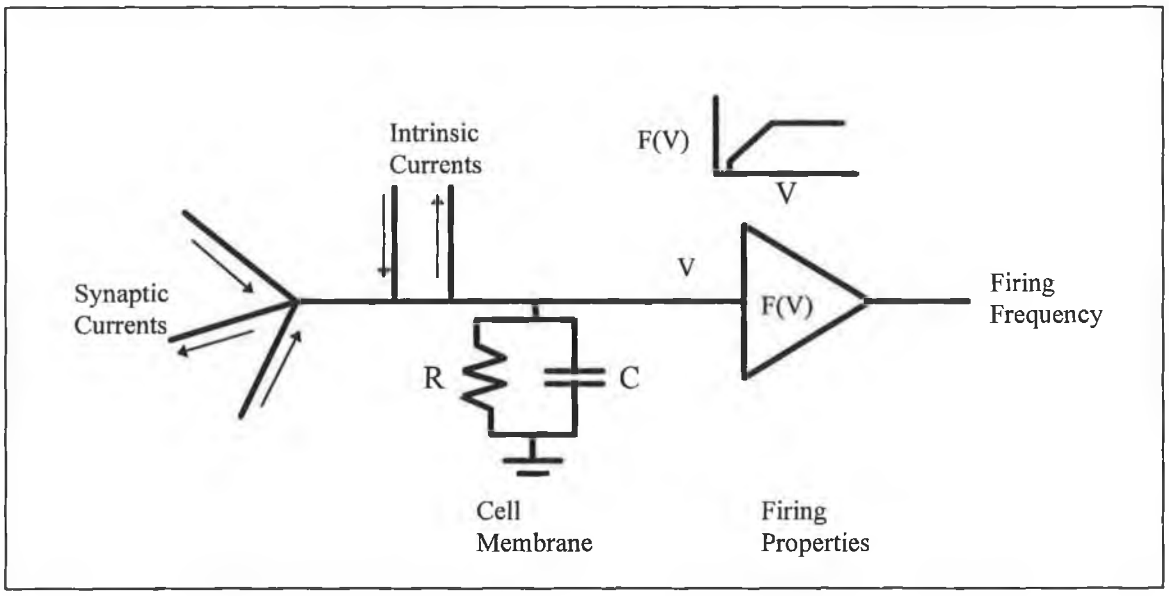


Figure 2.2 Neural Node structure implemented by Beer

Unfortunately, Beer's simulation ran as much as ten times slower than real time[2][p.63]. This meant that any attempt to directly recreate the same networks and behavioural patterns in real time would be unattainable using the existing hardware resources. This posed a large problem but the solution chosen was to simplify the neural node model. Hence, reducing the computational processing power required and thus allowing small networks to operate and be successfully updated in real time. This was necessary to achieve the first goal of the project which was to have a real time robot behaving in accordance with the initial behavioural levels of Brooks' subsumption architecture[4].

2.3.3 Application of Computational Neuroethology to this Project

The application of the computational neuroethology paradigm to this project was to model the construction of the nervous system of a simple animal such as an insect. It was decided to make the implementation as facile as possible by incorporating wheels into the robot structure rather than using mechanical legs. This, it was viewed, would allow the overall framework to concentrate on the sensory response characteristics of the robot. The generation of a mechanical gait controller was viewed as superfluous at this early stage of research.

The goal of this modeling was the production of a neural network which would control a robot in real time. The network, designed on a computational neuroethological basis, would be heterogeneous in nature. This meant that it would not be a physically fixed structure neural network form (as compared to a multi-layer perceptron network[31] for example). Also the individual neural nodes within this type of neural network have more than a single parameter governing their input/output behaviour. As Beer successfully demonstrated[2], this extra variability can mean that this form of network should potentially be able to display more complex functionality than a fixed structure, single variable neural network. This variability also allowed the use of fewer nodes and hence computation. This is very useful in an application concerned with real time operation and control such as robotics. Unfortunately it also means that none of the standard learning algorithms associated with existing neural network models are applicable. Hence each network must be designed manually.

This application of this type of neural network structure as a robot controller, as well as the real time advantages, seems to tie in very well, on an inspirational level, with Brooks idea of a subsumption architecture for the implementation of machine intelligence. However, the networks designed by Beer (which are based on existing neurobiological maps of small insects) only operated in the very strictly controlled conditions of a simulated environment within a computer simulation[2]. When this project was originally started, this computational neuroethological methodology for neural network design had not been used to successfully develop a real time robot controller dealing with a real environment. However, Beer does mention this particular application in the conclusion of his book[2].

As stated above, one of the main problems involved with using the work of Beer as it stood was that the simulation which implemented his neural networks for the control of his simulated insect operated very slowly. The simulation ran at a speed equivalent to three to ten times slower than real time[2][p.63]. For the purposes of this project the neural node model which Beer used had to be simplified (accepting the resultant degradation in an individual neural node's functional potential). This was in order to speed up the operation of the networks and allow real time operation. This is obviously essential in a real environment. This simplification was made doubly necessary as the networks produced in this project were to be run on a Motorola 'Force' board. This board used a MC68000 processor with a bus speed of only 8 MHz. The details of the simplification eventually used for the neural node are given in section 4.5.4.3.

2.4 Simple Genetic Algorithms

2.4.1 A definition of Simple Genetic Algorithms

The third academic source for the thesis is the work of David E. Goldberg[14]. Goldberg's work is in the area of *Genetic Algorithms.* Genetic Algorithms (GAs) can be used for finding a solution to problems in non-linear or complex problem spaces. GAs differ greatly from the traditional algorithms used for problem solving. GAs use a number of rules to 'find' an optimum solution to a given problem rather than derive a precise solution to a precise problem. They draw their inspiration from the apparent ability of the DNA structures contained in all living matter to solve problems in a gradual and optimum seeking manner.

The operation of genetic algorithms revolves around the use of parameters coded in string form, (which in the case of this project is in binary format). This string form is referred to as the genetic coding. GAs, in their simplest form, use constructive and destructive mutations of the genetic coding, a structured yet randomized information exchange between strings and a guiding 'objective' function in their search for a solution. The objective function guides the algorithm towards an optimal solution (and hopefully the optimum) in a given problem space. The optimal solution is evolved gradually as the algorithm 'traverses' the problem space. The location of the optimal solutions in a search is dependant on a number of parameters. These parameters include the availability of a smooth genetic search space, the correct setting of the genetic string's internal parameters (see section 4.3.3) and a well defined objective function.

Simple Genetic Algorithms (SGAs) are, as the name suggests, the most basic implementation of genetic algorithms. The term Simple Genetic Algorithm is used throughout this text because the code used to implement the genetic algorithm operation is based on the PASCAL code given in Goldberg's book [14][chpt.1]. This PASCAL code is called a simple genetic algorithm by Goldberg and the continued use of this term is purely for the sake of remaining consistent with the material in the reference text.

To prevent confusion the main terms and abbreviations used in this section and the remainder of the thesis in reference to simple genetic algorithms are now explained:

SGA	Simple Genetic Algorithm
Search Space	The problem space of the simple genetic algorithm.
Genotype	Bit string which encapsulates the parameter set of an individual.
Phenotype	Entity created from the decoding of a Genotype.
Individual	Refers to the Genotype and Phenotype as a single unit.
Fitness	Value assigned to individuals based on their performance used in reproduction of individuals.
Population	A collection of individuals.
Generation	A particular instance of a Population.

Figure 2.4 Terms used in reference to Simple Genetic Algorithms

2.4.2 What's new?

So what are the fundamental differences between SGAs and more traditional algorithms? Goldberg specifies four ways in which SGAs differ from traditional optimisation techniques.

1. SGAs work with a coding of the parameter set, not the parameters themselves.

2. SGAs search from a population of points, not a single point.
3. SGAs use payoff (objective function) information, not derivatives or other auxiliary information.
4. SGAs use probabilistic transition rules, not deterministic rules.

   [14][p.7].

SGAs require the natural parameter set of the optimisation problem to be encoded as a finite length string over some finite alphabet. For this project the coding is a binary string in order to minimise the effects of single mutations in the genetic coding. The precise coding used is described in section 4.3.3.

The operation of the SGA involves processing a number of strings representing a population of individuals. The search is carried out by using a structured yet randomised information exchange between the genotypes of a population. The purpose of the structured information exchange is optimisation of the average fitness of the population to find a single stable, optimal solution or individual.

2.4.3 Genetic Fitness

The genetic fitness of an individual is a number assigned to the individual based on its performance. The objective function in an SGA is usually responsible for the assignation of this number. This is done by evaluating and comparing the performance of individuals relative to some known, or unknown, optimal performance points in the overall search space. In the case of this project: each point in the search space represents a controlling neural network. An individual's fitness value can be compared, in natural selection terms, with the ability of an individual to survive and mate with another individual. The objective function is usually a function within the scope of the SGA program itself. However, for this project, due to the difficulty in defining what constitutes 'good' behaviour (see section 5.2), it falls upon a human tester to evaluate the performance and determine the fitness of the individuals.

2.4.4 Genetic Operators

This information exchange between individuals is implemented using a number of functions, the individuals' fitness values and what are referred to as genetic operators. In SGAs (as applied in this project) these are a reproduction operator, a crossover (or mating) operator and a mutation operator. The genetic operators are the basis of the operation of an SGA.

The reproduction operator is a process in which genotypes are copied according to the fitness of their respective individual's values. The higher the fitness of an individual, relative to the fitness of other individuals in the same generation, then the higher the probability of that individual contributing one or more offspring to the next generation.

The mutation operator is used to prevent the complete loss of important information. This can happen when the algorithm begins to converge towards an optimum. It is a probabilistic process which switches the value of a single bit location in a genotype, from a 1 to a 0 or vice versa for example. It allows the algorithm the possibility of retrieving an important bit configuration (or schema)[2] which may have been lost between generations.

Finally, the operation of the crossover operator is shown in figure 2.5 below. It is a process whereby the genotypes of two individuals, chosen by the reproduction operator, are mated and exchange information. The crossing site is chosen at random and can be at any point along the aligned strings. This means that simple reproduction without information exchange is possible (i.e. the cross site can be chosen at the end or the start of strings).

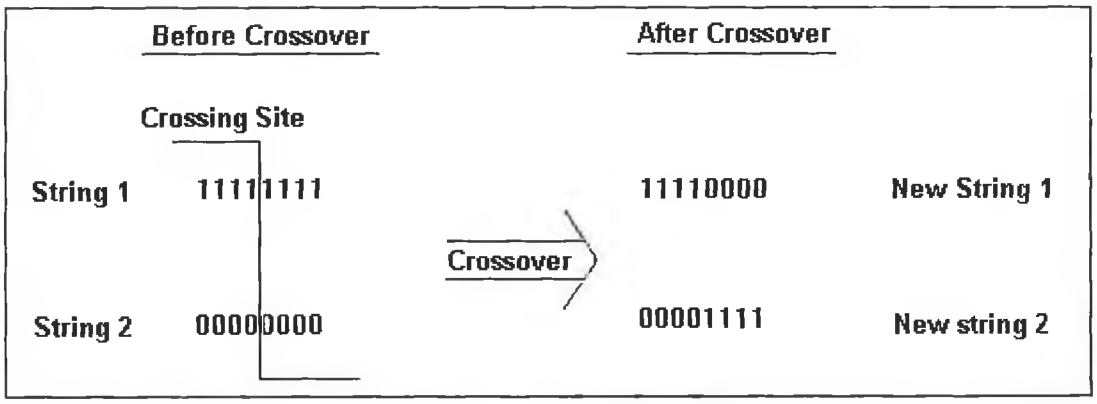


Figure 2.3 A schematic of simple crossover showing the alignment of two strings and the partial exchange of information, using a cross site chosen at random.

It is important to note that the SGA may not find a perfect solution to a given problem (it may not exist!). It strives only to improve on existing proposed solutions using the genetic operators described.

[2] A schema is a similarity template describing a subset of of strings with similarities at certain string positions. For a more complete description refer to Goldberg [14] or Holland [20].

2.4.5 Application of Simple Genetic Algorithms to this Project

The simple genetic algorithm is the most basic form of the genetic algorithm available. There are many others documented even within the texts already referred to [14,19]. However, I think that it is pertinent to re-emphasise that this thesis offers only a primary investigation into the area of the combined use of many different works and areas of expertise. That is why the majority of modifications made to existing works were simplifications (e.g. use of the SGA, the neural model used, the parameterisation of the neural model (see chapter 4)). The simplifications were used in order to attempt to obtain fundamental results which would verify the potential success based on the combination of the underlying processes.

For the purposes of this project; reproduction, crossover and mutation are the three genetic operators used. As stated above, the fitness of an individual network is evaluated by the network designer and not by an objective function within the program. Also the reproduction function does not allow generations to overlap. This was a decision made to simplify the implementation of the SGA.

Now that the structure for the algorithm's operation has been described, how does the genotype become a phenotype? This is achieved by the application of computational embryology.

2.5 Computational Embryology

2.5.1 Phenotype Growth.

Goldberg's work is being used in conjunction with the work of Richard Dawkins[9,10,11] to create neural networks for robotic control. In his work Dawkins uses genetic algorithms and a 'development' routine (which decodes the genotypes) to produce pictures on a computer screen that could be considered biological in form. He calls these pictures biomorphs and some do indeed resemble (in a two-dimensional sense) insects, some resemble trees and they can be made to produce a variety of 'biological' forms.

The pictures are generated using a development routine that decodes and uses the set of parameters encapsulated in the genotypes. These are parameters like: the number of times the recursive growth routine is called; the angle that branches or divisions in the pictures take, the length of the branches, etc. The choice of which individual is the 'best' or the most fit in a genetic algorithm sense is a purely arbitrary decision made by the user. In Dawkins' case this meant the reward of individuals who produced pictures that resembled something biological in nature.

For the purposes of this project the growth idea is adapted and used to grow the neural networks to control the robot. One difference lies in the fact that Dawkins does not use crossover in his application of the genetic algorithm. The crossover operator is used in the course of this project. It was hoped that by using the crossover operator that the SGA could come to an optimum more quickly than by depending on mutation alone.

The networks are grown from a decoding of growth parameters encoded in the genotype. The details of the genotype encoding and decoding are given in section 4.3.3.

2.6 Conclusion

In chapter 2 a general introduction to each of the main sections of the applied background theory was given. The concepts of subsumption architecture, computational neuroethology, computational embryology and genetic algorithms were introduced and some details of their application to this project were given. The specifics of their individual contributions to the project are detailed later in the thesis.

3. Hardware Implementation Details

3.1 Introduction

In this chapter, each of the relevant hardware components of the design framework will be described. The performance of each of the components will also be analysed and suggestions for improvements detailed where appropriate. The sections that will be described are the robot Single Board Computer (SBC) (section 3.3), the robot to SBC interface (section 3.4), the robot itself (section 3.5) and finally the simulation environment that the robot's behaviour was observed in (section 3.6). Firstly however, a general introduction to the physical hardware setup and connectivity will be presented in order to allow the reader to see where each part of the hardware is in respect to all the others. Overall, the hardware chosen for the implementation performed to varying degrees of success. The problems, successes and recommendations for improvements in the setup will be given at appropriate points within this chapter.

3.2 Hardware Environment Overview

In this section the overall hardware environment will be described in order to allow the reader to appreciate the positions of all the constituent hardware sections of the framework relative to each other. The hardware consists of a number of very distinct parts, each of which is responsible for a number of different areas of the overall operation. A pictorial description of the environment is shown in figure 3.1.

As can be seen from the text in the picture, a large quantity of processing for the overall framework is done on the PC. The PC is responsible for both the implementation of the simple genetic algorithm and for the implementation of the artificial neural network development routines. The PC also controls the generation of the network simulation executable code which is transmitted to the SBC in MC68000 microprocessor assembly code.

The format of the assembly code transfer is S19 format[30]. The transferable code is generated using the PARAGON C Cross Compiler[28] software which generates the MC68000 assembly code from the source code written in the C programming language.

The human user also uses the PC for controlling the input of behavioural scores. Each of the above software operations are detailed in chapter 4. The type of PC used varied over the course of the project from an 80386SX personal computer to a Pentium processor

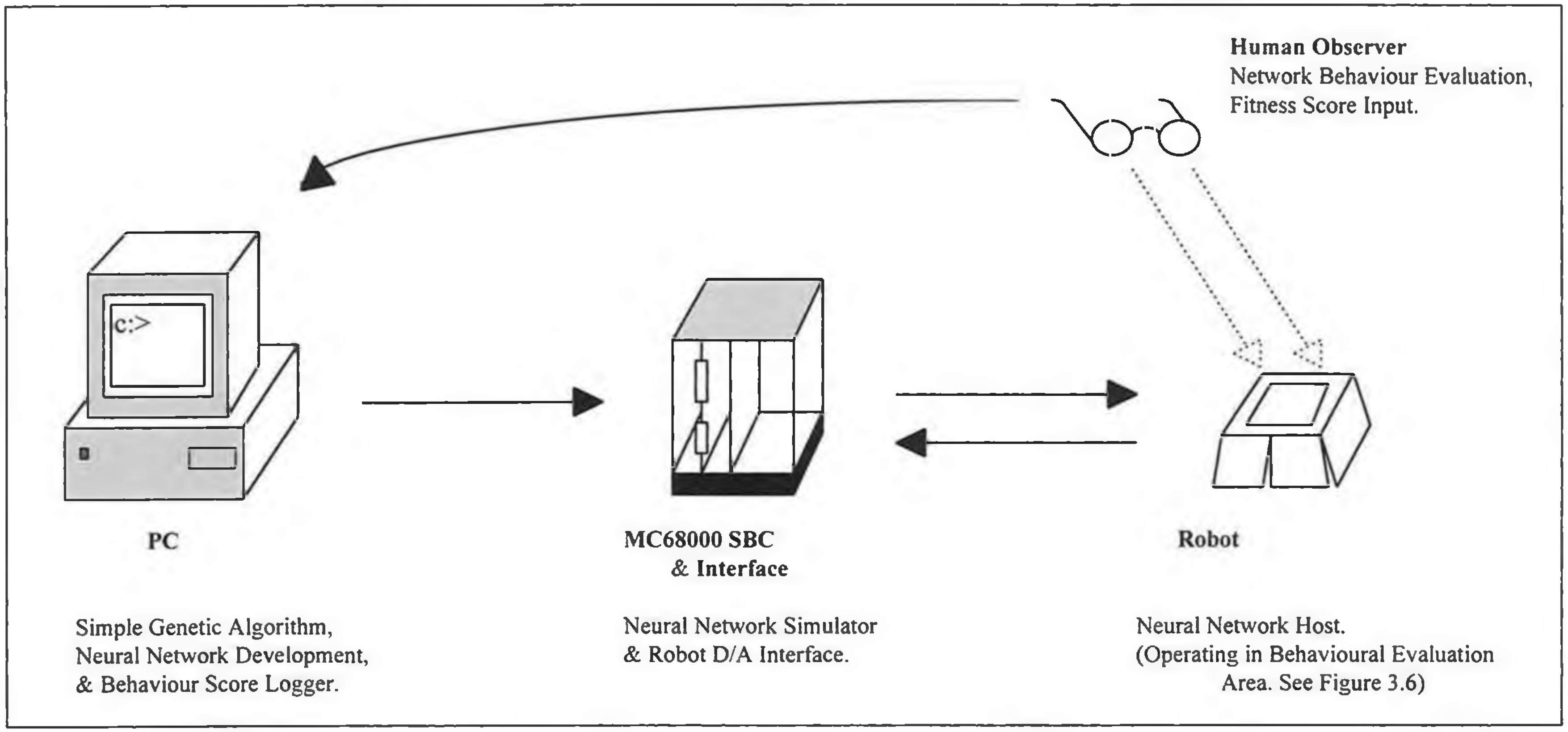


Figure 3.1 Graphical Description of Hardware Environment

personal computer. The 386 was more than adequate for implementation of the overall framework.

However, the increase in power was the direct result of the combination of a desire for greater support for ancillary operations in the project (such as word-processing), and the desire to decrease the software compilation times during development. Also, the improved PC performance increased the general operation speed during neural network evaluation runs. The evaluation of each network took approximately 4 minutes from beginning to end and any increase in speed was a great advantage in simply preventing boredom as each generation of robotic behaviour was evaluated. This was actually quite an important issue as maintaining concentration was very important in evaluating the performance of the robot objectively.

The SBC is responsible for executing the software which implements the neural network controlling the robot. Its sole responsibility is to run the software which reads the robots sensory input and controls the corresponding motor output. The SBC uses an MC68000 microprocessor.

Two different SBCs were used over the course of the project. The first was a MOTOROLA MC68000 Educational Computer Board (ECB). However, the use of the ECB was not satisfactory as it suffered significant hardware failure twice. The second time this happened was at a crucial point in the project and the failure forced the project to be delayed by about 6 weeks. In total, because of the attempt to continue to use this board by waiting for replacement parts, over two months delay occurred before testing and evaluation of the overall framework could be carried out. This only became possible when it became obvious that it would be impossible to delay the project any further. Eventually, a different SBC was selected, allowing work to continue. This new board was also MC68000 microprocessor controlled but it had the added, and significant, advantages of having both an 8 MHz microprocessor and a more reliable power source. The previous SBC had only a 4 MHz microprocessor and significant problems with its power source. These difficulties and their solutions are detailed in section 3.3 which also discusses other issues pertinent to the SBC.

The next piece of hardware is the interface between the SBC and the robot. It acts as a digital to analog converter for the robot motor output signals generated by neural network software running on the SBC. It uses two power amplifiers to boost the power of this converted output in order to drive the motors. It also passes the sensory information, generated by the robot, to the SBC. The interface went through a number of significant changes throughout the project. These changes are detailed in section 3.4 along with details of the final design of the board.

The final piece of hardware is the robot itself. The robot was constructed from Technic LEGO® building blocks. It was 'inherited' from a previous project in the School of Electronic Engineering and was redesigned and enhanced. A number of problems still remain with the robot's construction and recommendations for its future enhancement are detailed in section 3.5.3 along with the design as it stood at the end of the project. The robot operates in a closed physical environment which is described in section 3.6.

Overseeing the operation of all this and responsible for the evaluation of the robot's behaviour in the real world is a human observer. The observer is responsible for synchronising the overall evaluation process and inputs the scores assigned to the robot's behaviour into the PC. The details of this evaluation are given in chapter 5 .

3.3 Robot Single Board Computer (SBC) Details

3.3.1 Introduction

In this section the robot SBC which was responsible for the implementation of the neural network simulator software (see section 4.5) will be detailed. The responsibilities of the SBC will be detailed and a description of the changes undergone in its physical configuration over the course of the project will be given. Also, the manner in which the SBC communicated with the robot interface (see section 3.4.3) and the PC will be described (see section 3.4.3). The difficulties encountered with the SBC, mentioned in section 3.1, over the course of the implementation will be described. The reasoning behind the choice of hardware will be given and recommendations for future enhancements will be explored.

3.3.2 Aspirations

The neural networks designed on the PC were each to be tested on the robot. When this was decided, early in the project, the question of what the networks would be run on was broached. A long term decision was made concerning this which on reflection may seem a little optimistic in its aspirations. It was decided to execute the neural network simulator software on a dedicated SBC rather than from the PC itself. The decision to use a dedicated control board was made because it was hoped that, eventually, the robot would become a single unit incorporating physical structure, power source and microprocessor control board. It was hoped that this may even have been possible within the time frame of this project. However, a significant degree of disruption occurred which slowed the progress of the project. These difficulties will now be detailed.

The original SBC used was a Motorola MC68000 Educational Computer Board (ECB) [30]. The ECB had been used for a number of years within the School of Electronic Engineering and was apparently quite old. It did not use a dedicated power supply and was powered instead using a MINILAB power unit from the School of Electronic Engineering laboratories and the first problem that occurred was in this power supply setup. The power supply problem manifested itself by causing the SBC to periodically reset itself. This caused more frustration than damage as all that was required was to download the network simulator software to the SBC again. However, as every aspect of the framework testing was quite time-intensive any delay was extremely discouraging and disrupting.

The second, and more major problem that occurred was discovered to be due to, after post failure analysis, to the physical condition of the original board used. Although the ECB may have sufficed for a shorter term project it seemed unable to satisfy the operational requirements it was under. Subsequently the ECB suffered two hardware failures. These failures may not necessarily have been due directly to the power source problems mentioned above (no *detailed* examination was carried out after the failures) but they may have been related. No detailed examination was carried out because the SBC being used was simply a tool for the project. It was perceived that it would have been detrimental to the progress of the overall project to spend excessive time examining the hardware failures.

The second of the hardware failures occurred at a point where all the constituent parts of the framework were in a position to be tested together. The delay caused by this failure was in excess of six weeks during which time all work effectively ceased as no testing could be carried out on the work already done. The board was sent away to be repaired but unfortunately the repair was not economical. Therefore, the ECB was replaced with a different SBC.

This new SBC was much more successful and also stable. This may have been as a result of the fact that it had a dedicated power supply. The new board was also obtained from within the School of Electronic Engineering and was a Motorola FORCE Board. As well as this dedicated and stable power source, the FORCE board also had an 8 MHz processor speed which was an added advantage as the ECB only had a 4 MHz processor speed. Any speed that could be gleaned at all from the setup was viewed, correctly I feel, as an advantage to the real time aspect of the framework and its testing.

3.3.3 Operation of the SBC

As mentioned previously, the SBC was responsible for the execution of the neural network simulator software. This neural network software was downloaded in assembly language format to the SBC, each time the SBC was turned on, in order to execute it. This was necessary as the SBC could not store any user information after powerdown. The software was transmitted from the PC using the KERMIT software transport protocol[30]. The download situation was not as complicated as it may have been because the SBC had a resident control system in its ROM which facilitated the relatively pain-free receipt of software transmitted from an external source. The software on the SBC could also be executed from the remote source. This facilitated the implementation of a batch routine on the PC which automated the transmission of the neural network simulator software as well as the neural networks produced by the design framework.

On the other side of the operation, the SBC communicated with the robot interface using a Parallel Interface/Timer (PI/T) chip that was resident on the SBC. This chip transmits the motor outputs in a digital format to the interface board. This format is an eight bit binary number where the most significant 4 bits and the least significant 4 bits corresponded to each of the motors. The motors are bi-directional, to allow the robot the ability to reverse. The code relating the speed and direction of each of the motors is as described in figure 3.2 below.

Binary Code	**Motor Output**
0000	Full Reverse
...	...
0100	Half Speed Reverse
...	...
1000	Full Stop
...	...
1100	Half Speed Forward
...	...
1111	Full Speed Forward

Figure 3.2 Robot Motor Output Coding

The PI/T also controls the receipt of the sensory information from the robot through 6 of its channels. The PI/T allows expansion to allow up to 16 I/O bits as it has three 8 bit ports allowing bi-directional I/O.

3.3.4 Summary

In summary, the problems caused by the choice of SBC, although this choice was well intentioned, caused a significant amount of disruption and delay. In retrospect it may have been more sensible to have used the PC's serial I/O ports to communicate directly to the robot interface. Although this would have lead to a significantly more complex robot/controlling computer interface, this configuration would have been a lot more stable at the early stages. This stability would have due directly to the absence of the SBC link in the network design to network operation chain. This was the link that caused so many of the early difficulties in the project. This stability could potentially have allowed a lot more to be achieved in the time scale of this project. However, in opposition to this argument, it should be noted that the aspirations of the project, at inception, included the development of an independent robot incorporating the robot, the SBC and a power unit.

Certainly, there is no argument against the preference of designing the neural networks to operate on a SBC similar to the one used given the aspirations detailed above. Of course, a more powerful processor could be used which would further facilitate the real time operation of the robot. Choosing such a board allows an easier transition, whenever it would become feasible, to the development of a more autonomous robot. The addition of a power source and more powerful motors and stable physical framework to support the SBC would be all that would be required. That said however, the feeling remains that more may have been achieved if the PC was used exclusively at the early stages of the project due simply to the inherent stability of most PCs and the ease with which they can be controlled. This stability would have prevented the difficulties that were overcome. This would have provided more time for the analysis and improvement of other aspects of the project and project such as the implementation details of the SGA and the computational embryology software.

3.4 Robot/SBC Interface

3.4.1 Introduction

In this section the interface between the physical robot and the controlling SBC will be described. Its origin will be detailed as it was designed before this thesis was conceived. Its final design and the transitions it underwent will also be detailed in this section. Finally, any recommendations for its further improvement will be detailed.

3.4.2 Beginnings

The basic interface board design and construction was originally inherited from a final year project in the School of Electronic Engineering in Dublin City University [21]. The state of the interface board on receipt was not very healthy. The interface had been designed but had not been completed. The board was in wire-wrapped rather than Printed Circuit Board (PCB) format. A number of the connections were quite loose when the board was first tested for its use in this project. Also, the board did not use any stabilising capacitors on its power supply which produced power spikes that may have been jointly responsible for the failures of the first SBC used (see section 3.3). Initially the board was simply repaired while the other more critical areas of the design framework were being worked upon.

Eventually however, it became obvious that it would be necessary to redesign and reconstruct the interface in order to enhance its reliability - thus allowing its continued used over the extended periods involved in neural network testing. Time would have been saved if the board had simply been redesigned from scratch.

3.4.3 Operation and Description

The redesign of the interface involved primarily the simplification of its design but it also involved the improvement of some of its physical construction characteristics to improve reliability. One of the most obvious redesigns was in the reduction in the number of Operational Amplifiers (Op-Amps) on the interface board. The number of Op-Amps was reduced from four to two. Originally the board used two Op-Amps per channel (i.e. two for each robot motor). The first Op-Amp in each channel decoded the four bit string which contained the motors' direction and speed as per figure 3.2. This configuration was a simple summing amplifier configuration as shown in figure 3.3. 0utput from the summing amplifier was then passed to a power amplifier (Power-Amp) circuit in order to boost the output to a level powerful enough to drive the motors on the actual robot.

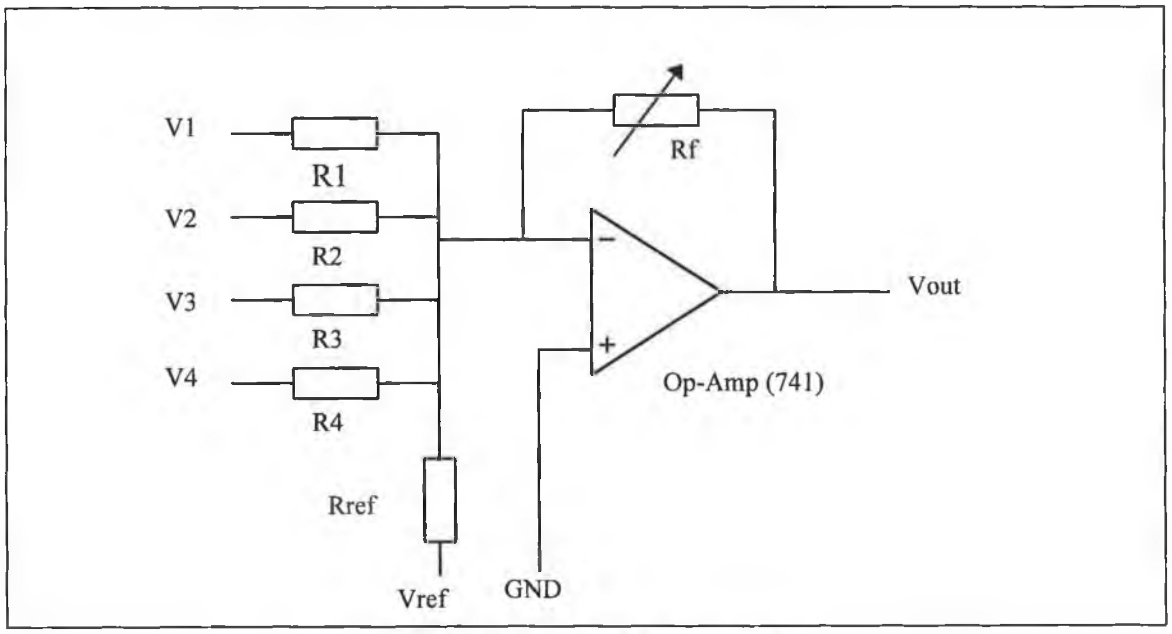


Figure 3.3 Summing Amplifier Construction

The reduction in Op-amps was implemented by setting up the summing amplifier across the Power-Amp and removing the dedicated summing amplifier circuit across the ordinary Op-Amp completely. The set up for the board was thus reduced in complexity as well as reducing the number of connections (and thus the possibility of hardware failure). The calculations involved in designing the summing amplifier configuration as well as the final design of the interface itself are shown in Appendix D.

The original interface was also populated largely with potentiometers. These were replaced with fixed resistors in order to increase the stability of the interface board. This was because the potentiometers used in the original design were not of a very high quality and they appeared, on examination, to have been damaged. This may have been due to the fact that the board was not stored in any form of protective box. The potentiometers used seemed to demonstrate a tendency to short-circuit themselves quite regularly. This obviously affected the operation of the interface quite adversely by changing the apparent motor responses to the input stimulus applied to the robot. This situation made what were apparently identical neural networks behave in totally different ways to the same stimulus and this uncertainty would have made it impossible to effectively evaluate the behaviour of the robot later in the project.

It was also mentioned in the introduction to this section that the interface board was of a wire-wrap configuration which was tidied up at the start of the project implementation. As time progressed however it became obvious that this was unacceptable. Prompted by the continual loosening of the connections, especially when coupled with the problems associated with the potentiometers of the original design, it was decided to transfer the circuit to a different and more stable medium. It was decided to used a vero-board construction. This allowed a speedy construction and with the constituent parts being soldered into place the entire construction is much more stable and physically secure.

Also the interface was improved by added a smoothing capacitor circuit to the power input (see Appendix D).

3.4.4 Summary

After discovering the failings of the original interface and its subsequent redesign and reconstruction based on those findings the interface was a very stable and reliable member of the hardware setup. The difficulties that were previously encountered did not resurface again during the final stages of the project.

An advancement that could perhaps be made would be to finalise the design by committing the interface circuit to a PCB format. However that would only really become a consideration were the robot itself to become autonomous in the manner suggested at the end of the section on the robot SBC. Obviously, the integration of the SBC with the robot could not happen if the interface were not also integrated. Similarly, integration of the interface alone would not be a very sensible step as it would not be any advantage to the operation of the robot itself.

3.5 The Robot

3.5.1 Groundings

As already mentioned already, the robot plays a vital role in testing and verification of the neural networks grown using the phenotype development software (see section 4.4). Also the testing of every change and every improvement on the physical robot before advancement is at the crux of Brooks subsumption architecture approach, which the project is attempting to implement. At the same time however, due the experimental nature of the project, it would have been foolish to have designed a very sophisticated robotic structure. It was rather more necessary to design a robot that would be physically flexible. It was impossible to predict exactly what problems could have been encountered, problems which could potentially have forced a redesign.

It was hoped that any minor imbalances, such as a slight difference in motor characteristics, could be compensated for by the neural network operation. In a sense, hoping that a certain level of self-awareness would become apparent. The structure, in particular the sensory bumper frame, did not appear to affect the mobility and operation of the robot in its environment excessively, although a number of small problems were encountered. These difficulties and their potential solution are detailed in section 3.5.

3.5.2 Description

The robot consists of a number of parts which can be seen in figure 3.4. The individual parts are the motors for the wheels, the micro-switches that act as sensors for the robot, the sensor bumper frame to increase the activation area of the micro-switches and the physical robot structure itself to hold it all together. The final part of the robot is the cable connection between it and the interface board connecting it to the SBC.

The original robot parts were inherited from another project[21] run in the School of Electronic Engineering in Dublin City University, and its basic design, which can be seen in figure 3.4, remained largely unchanged over the course of the project. The robot was constructed using constituent parts of a 'LEGO Technics' building blocks package. This offered the ability to change system design quickly as well as allowing a prototype robot to be built in minutes rather than weeks.

A number of enhancements were made to the inherited robot because, as the project progressed, a number of problems became apparent. The first problem detected was that the micro switch sensors used on the inherited robot were not sensitive enough to be useful. The operating force required to close an individual switch was equal in magnitude to the force required to move the robot physically from one position to another. The sensors were thus replaced with micro-switches of a much lower operating force. Also, they were glued to the robot framework to prevent them from slipping out of position. Previously the sensors were attached using a small quantity of industrial tape and under continued use they would slip from position. The replacement was successful and the new sensors did not suffer the same problems as those that were replaced.

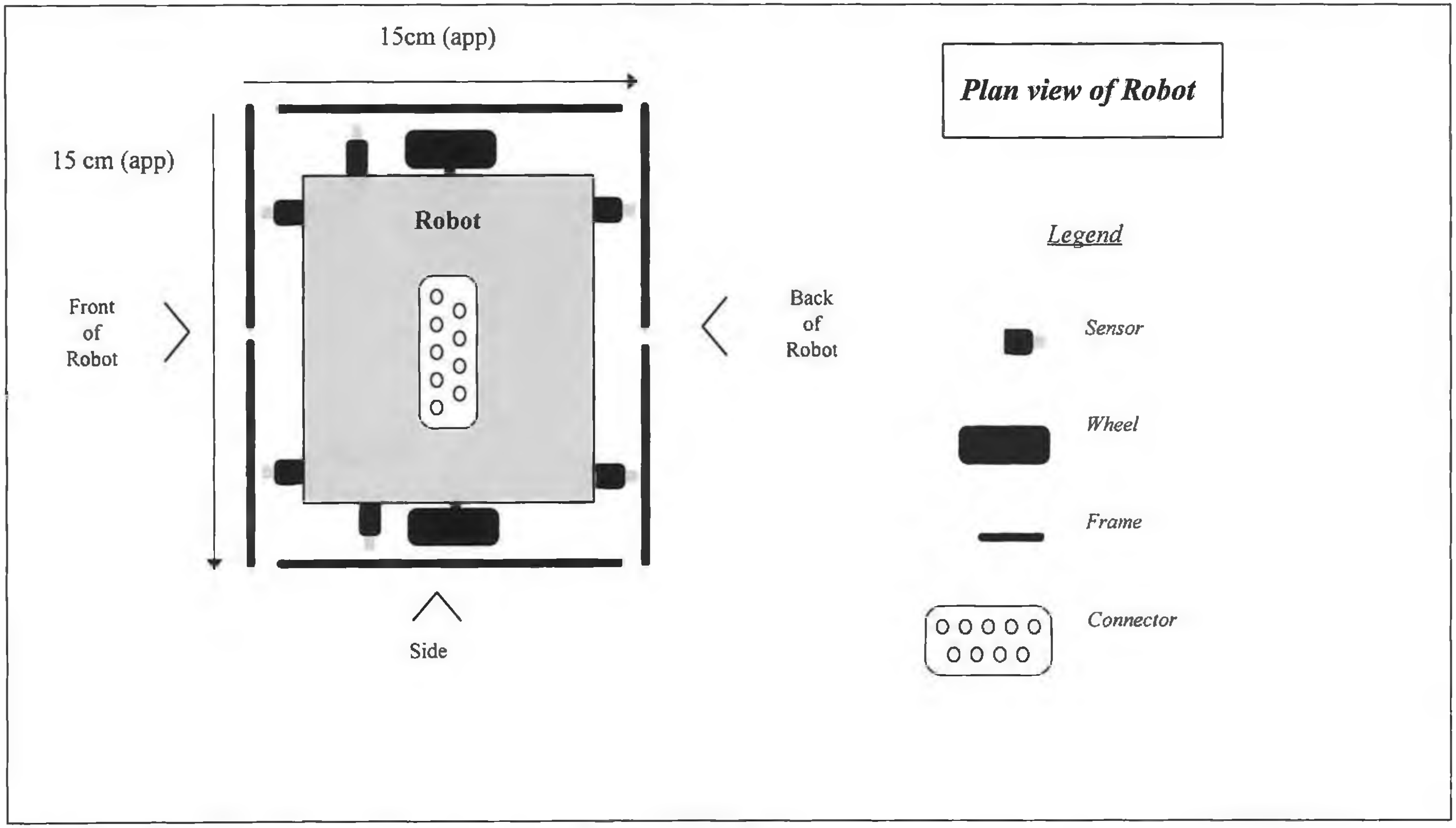


Figure 3.4 The Robot

The cable connecting the robot to the SBC via the digital to analog interface had to be replaced. The existing cable was unsatisfactory as it consisted of a number of individual wires and was connected permanently to the robot. This connection's soldering was exposed and was prone to breaking. This cable was thus replaced by a 16 way ribbon cable. Sixteen channels was more than what was required for the level of information transfer between the robot and the SBC. However, the use of different sensor types, or even expansion to the use of more than the existing six micro switch sensors already in place, was being considered. The new cable was terminated by a 16 way D-connector. This was advantageous as it meant that it could be disconnected from the robot when in storage. The D-connector also reduced the risk of any damage which may have occurred due to any twisting of the cable.

Next, a sensory bumper frame was constructed for the robot to increase the area that each sensor covered. A frame of the shape shown in figure 3.4 was designed. This shape was chosen because it is easy to construct and allowed continued operation with the existent sensory layout on the robot. The frame was constructed using hard cardboard allowing it to be replaced easily if it should become damaged. The frame worked well in increasing the area that could activate the sensors.

3.5.3 Summary

The robot satisfied its requirements physically. Its simple design was ideal for the early stages of a project such as this. The use of LEGO building blocks in particular was very successful in building the prototype. In the longer term however, a more robust robot would necessarily be built. LEGO building blocks have a tendency to separate over time. This affect caused the sensory framework to distort and become misshapen over time thus preventing it from operating correctly until repaired. The largest problem with this was that, as the LEGO separated and the frame became distorted, the edges of the sensory frame flaps would occasionally become trapped against some corners of the evaluation environment. This occurred particularly when the robot was performing some form of circular motion (e.g. on reflex reversal from a wall while edge following). The robot would then have to be freed manually and the frame adjusted. To improve the framework and preventing a similar situation occurring a more comprehensive, flexible and circular framework should be constructed in a form similar to that indicated in figure 3.5.

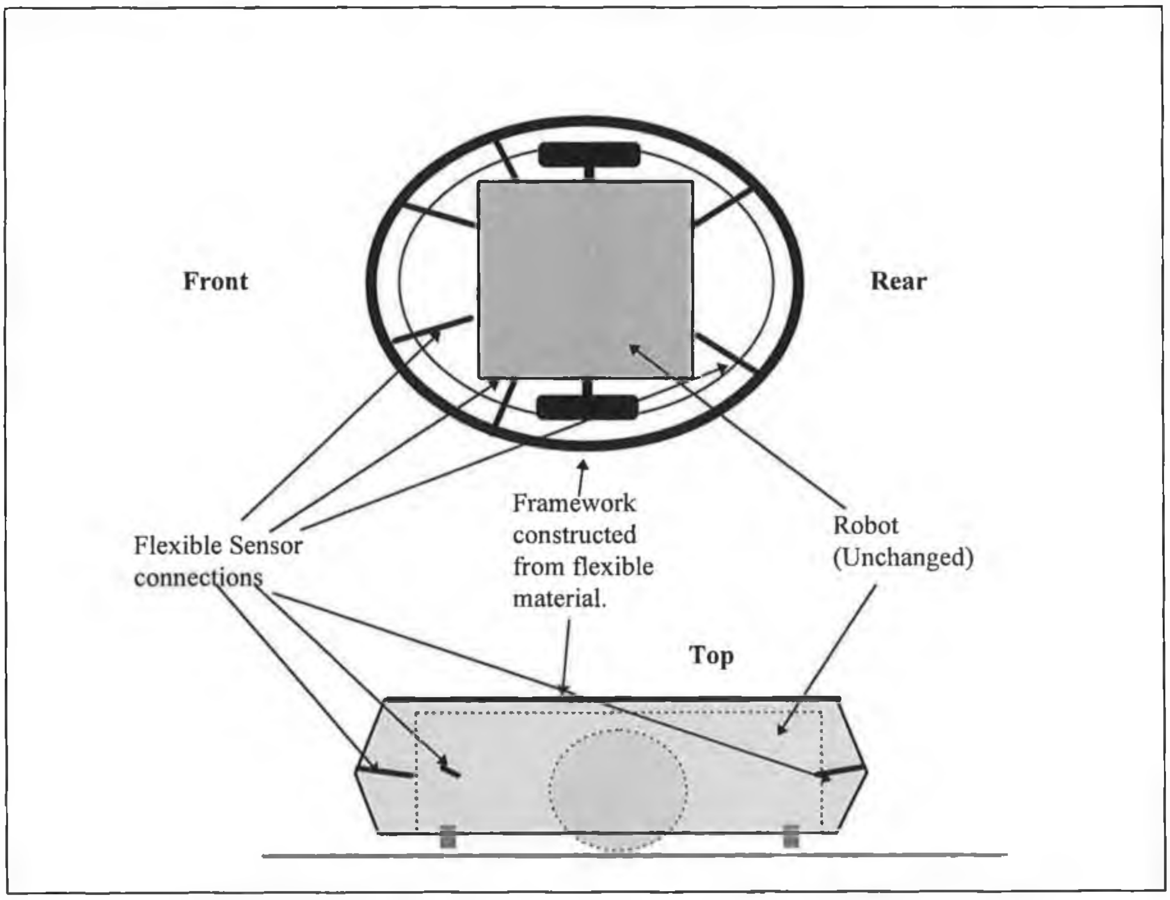

Front
Rear
Flexible Sensor connections
Framework constructed from flexible material.
Robot (Unchanged)
Top

Figure 3.5 Improved Sensory Framework

This new framework would prevent many of the physical difficulties described and also provide a more comprehensive and operational sensory area.

A further beneficial enhancement would be the inclusion of a motion detector[3] in the sensor array. This would become useful when the robot becomes stuck in its environment. The design of the controlling neural network could reward the generation of random behavioural response when the robot becomes stuck. The motion detector could operate in a number of ways, but the easiest would be to include some form of current detector on the wheels' motors. When the motors are prevented from turning, (e.g. when the robot becomes stuck), the internal motor resistance is increased which forces the current input to be increased. This detector could be incorporated onto the robot/SBC interface board design.

[3] Some form of motor torque detector perhaps.

3.6 Behavioural Evaluation Environment.

3.6.1 Introduction

In this section a simple description of the environment in which the robot's behaviour was evaluated will be given. The environment was a very simplistic one which was designed to test the ability of the networks generated to control the robot to prevent the robot becoming stuck.

3.6.2 Description

The environment, as mentioned in the introduction to this section, was designed to test the ability of the networks generated to satisfy the first level of Brooks' subsumption architecture specification[4] . Namely, the desired behaviour of the robot was to move around the environment and not become stuck. The environment is shown in figure 3.6.

The environment walls were constructed from LEGO building blocks. The environment was 1.2m by 0.9m in size. A number of simple challenges were designed into the environment to test the behavioural success of the robot in the environment. These were as follows. Firstly a large open space to see if the robot would simply spin in circles if there was nothing impeding its path (A). Secondly, a narrow channel was constructed to see if the robot could reverse itself out such a gap, if it got itself into it(B). A couple of 'rooms' were constructed with only a single exit. The first of these was a large room with a narrow exit. (C) The second was a smaller room with a single exit (D) and finally a small room with an angled wall and a narrower exit(E).

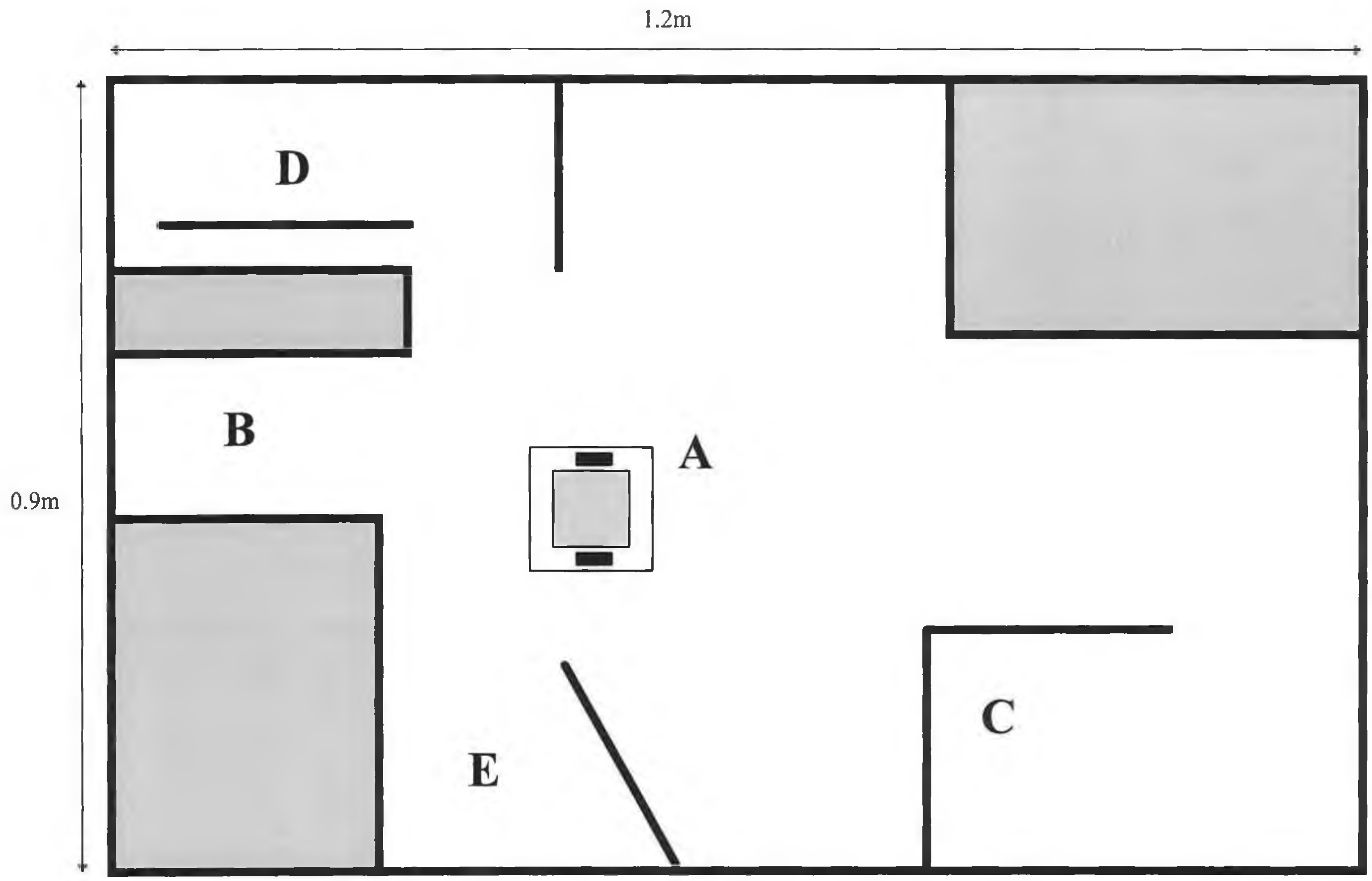


Figure 3.6 Behaviour Evaluation Environment

3.7 Summary

The hardware used in the course of the project caused some significant problems which were beyond the scope of this project to analyse fully. Rather it was more essential to simply identify the causes of any such difficulties and come up with the most rapid and effective solution to allow the project to continue. The failure of the original SBC used is a prime example of such a situation. In general however, the hardware caused few problems due to the simplicity of the overall design. The hardware, although essential from the point of view of the project's objectives, was essentially secondary with respect to the theoretical aspects of the project. Certainly, any further exploration of the area would require a significant increase in robustness and functionality (from the robot in particular), but for a prototype framework, the hardware components were essentially satisfactory. Any difficulties that did arise were dealt with quickly and efficiently allowing the project work to continue without overly significant hindrances.

4. Software Implementation Details

4.1 Introduction

In this chapter all the relevant details, and where appropriate an analysis, of the software components of the project will be presented. This includes descriptions of the overall software environment (see section 4.1), source code and executable code considerations (section 4.2). It also documents the implementation of the simple genetic algorithm (section 4.3), the neural network growth software(section 4.4) and the neural network simulator (section 4.5). Finally an overall summary of the software written (section 4.6) is given. Also, recommendations for future improvements will be given at appropriate points in the chapter.

The types of software used in the development of the experimental framework were very varied. It consisted of high level code controlling the implementation of algorithms based on existing natural systems. Assembly level functional code was used to control the i/o ports for robotic control. Software was also used to implement remote machine operation controllers using existing applications and a dedicated batch process. This variability in software type was due purely to the diversity of functionality that was required by each section of the hardware and the overall framework.

The main language that was used in the software development was C. This was for a number of reasons. The first and most primary reason for this choice was that the C language supports the implementation of both high and low-level code. This allowed the relatively simple implementation of all the higher level algorithms. It also supports the programming requisites for bit and byte level manipulation of strings. This bit-level manipulation is visible in the i/o port configuration settings and readings on the robot SBC as well as in the string manipulation of the genotypes used by the simple genetic algorithm. The secondary reason for the use of the C language was my own familiarity with the language. This familiarity applied both to the development platforms used as well as with the level of software control that can be afforded by the C language structure.

The development platforms used in the development of the C source code were the 'Turbo C++' compiler/development platform and the Borland C++ compiler/development platform packages. All the programs written were in standard C format and did not exploit the capabilities of object oriented programming techniques. The C language was used in development of the SGA, computational embryology, computational neuroethology (neural network simulator) and artificial neural network simulation software.

The secondary programming language that was used in the design framework was MC68000 assembly language. This was the native language of the robot SBC. Therefore, the neural network simulator software, which was written in C for the reasons outlined above, needed to be converted to the 68000 assembly code for implementation. This was achieved using the PARAGON C cross compiler package whose operation is described in section 4.2.

A number of difficulties presented themselves over the course of the project which had significant effects on the programs as the development continued. These difficulties will be documented in more detail later in this chapter. The problems were all overcome however and, by the end of the project research period, the software being used was both robust and reliable. Changes could of course be made to the final software to improve both the performance and the functionality and some recommendations for those changes are made in the summary of this chapter. However, I think that the results obtained through the use of this software vindicates the choices and decisions made as it was developed. These decisions and the final state of the software used will now be described.

4.2 *Paragon Cross compiler*

4.2.1 Introduction

The PARAGON C Cross Compiler [28] was used to convert the source code, written in C, to the MC68000 assembly language. This conversion was necessary for execution of the network simulator code on the robot SBC. The PARAGON cross compiler supported the fundamentals of the C language. However the compiler caused some difficulties as the differences between itself and the C compilation platforms used (see section 4.2.2) for the source code generation became apparent. One of the areas in which the differences can become apparent is in the manner in which the compilers handle cross data type operations. This is because different compilers have different conversion routines to deal with these situations. To illustrate one of these situations, consider what happens when a real number is multiplied by an integer. Does the integer become converted to a real number before calculation or it the real number converted to an integer first? Over time however, these discrepancies were ironed out and the PARAGON product proved itself able to satisfy the requirements made of it very successfully.

4.2.2 Operation

The operation of the PARAGON C cross compiler was very simple. The software written in C was simply used as an input. This software was developed and decoded using a network simulat*ion* software package. The differences between the simulator and simulation are described in section 4.5. For now, it is sufficient to know that the simulation used the same core software as the simulator but also included a section that mimicked the motor outputs and sensory inputs of the robot. Only when the simulator code was at a point where it could be tested on the physical framework of robot and SBC was the PARAGON C cross compiler was used.

The PARAGON Compiler software consisted of two separate programs. The first cross compiled the C software and the second program assembled and linked it. The resultant software was then downloaded to the FORCE SBC via the serial port of the PC using the KERMIT software[29].

The cross compiler was a command line compiler which used a configuration file (see Appendix A) to generate the assembly level code for the robot's MC68000 FORCE SBC. This configuration file was used to specify the areas of memory that could be used for the executable code on the FORCE SBC. This configuration file also allowed the specification of the format of output that could be produced by the compiler. This included the production of memory mapping reports as well as full assembly listings and S19 format file output[29].

4.2.3 Summary

The PARAGON C Cross Compiler software fulfilled all its functional requirements admirably. Although not very pretty or easy to use, in comparison to the Turbo C++ development platform for example, it performed admirably. One advantage it had was that the cross compilation procedure, once it was settled, could be operated using a single command on the PC by using batch files. This meant that it simply did its job with very little fuss, thus leaving the developer to concentrate on the code being written rather than where it had to work.

4.3 Simple Genetic Algorithm Software

In this section of the thesis the application of Simple Genetic Algorithms (SGAs) to neural network design will be detailed. More information regarding the operation of an SGA and the genetic operators used in its operation can be obtained from section 2.4.

At the crux of genetic algorithm operation lie what are called genetic operators. There are three genetic operators used in the SGA. These operators are explained and described in section 4.3.5. Some of the C code which implements the genetic operators is listed in this section.

There is also a section on the random number generator being used in the SGA (section 4.3.4).

Section 4.3.1 gives a brief re-introduction to SGAs and also deals with the reasoning behind the choice of simple genetic algorithms for the development of neural networks. How the basic SGA, outlined in Goldberg [14] (and in section 2.4), has been modified for this project using the work of Richard Dawkins[9,10,11] will also be detailed. Dawkins' work is treated in more detail in section 4.4.

4.3.1 Introduction and Reasoning

For convenience, the table outlining the terminology used in this section is reproduced for the reader.

SGA	Simple Genetic Algorithm
Search Space	The problem space of the simple genetic algorithm.
Genotype	Bit string which encapsulates the parameter set of an individual.
Phenotype	Entity created from the decoding of a Genotype. For this application the phenotype is a neural network.
Individual	Refers to the Genotype and Phenotype as a single unit.
Fitness	Value assigned to individuals based on their performance. Used in reproduction of individuals.
Population	A collection of individuals.
Generation	A particular instance of a Population.

Figure 4.1 Terms used in reference to Simple Genetic Algorithms

As discussed in section 2.4, the simple genetic algorithm is an optimising search technique. It has been shown to be a useful tool in the traversal of non-linear or complex search spaces, particularly in applications that do not necessarily require a precise solution, but do require a solution *approaching* the optimum. A neural network of the form being used in this project is one such application. Therefore, its implementation within the design framework to guide the development of a neural network to control the robot is ideal.

Also, the simple genetic algorithm is based on a natural system. It draws its inspiration from the apparent ability of real genetic material, in the form of DNA, to adapt to its environment. As stated in the thesis introduction, one of the aims of this project was to model natural systems as much as possible, within the confines of the project, to achieve its objective. The use of the SGA satisfied this criterion.

The application of the SGA is based on the PASCAL code which is documented in Goldberg[14]. This allowed the algorithm to be applied very quickly and there was no room for error in the translation from PASCAL code to C code simply because the PASCAL code was very straightforward. There was only one major difference between the SGA as laid out in Goldberg[14] and how it was applied to the development of neural networks in this project. The difference lies in the fact that no explicit objective function[4] is used within the course of this project. Instead, fitness values are assigned to the individuals by the network designer based on the rules documented later in this document (see chapter 5).

Allowing the designer to assign the fitness values rather than using an explicit evaluation/objective function resembles the work of Richard Dawkins[9,10,11]. Dawkins used just such a system to guide the development of his 'biomorphs'[9]. However, for this application it is not the shape of the pictures produced which is rewarded but the ability of the neural networks produced to control a robot. Dawkins' work and its application is described more in section 2.5 and in section 4.4.

[4] See section 5.1.

The technical application of SGAs within the framework of neural network development will now be outlined.

4.3.2 Application

The simple genetic algorithm works using a generation of genotypes. The greater the number of genotypes, then the greater the chance that one will 'fall' close to a local optimum in the search space. This hopefully will reduce the amount of time necessary to find a suitable optimum or solution in the search space. However, due to the time taken to evaluate each genotype, it is necessary to reduce the number of genotypes to a manageable amount. This figure must also reflect the size of the search space. For this project the number chosen was initially ten in order to test the operation of the SGA programs. However, when the program was used within the design framework this number was increased to twenty in order to increase the possibility of discovering a genotype approaching the optimum in the search space.

The twenty genotypes were generated by simulating the toss of an unbiased coin. This produced the first random generation of genotypes. The random nature of the first generation distributes the SGA search's starting randomly across the genetic search space.

The genotypes themselves are binary strings of 54 bits. Different sections of the genotype contain the coded parameters that are used to grow the neural networks. The details of this growth are given in section 4.4.3. As described in section 2.4 of this project, SGAs operate on a coding[5] of a systems parameter set and not the parameters themselves. The specific coding of each genotype used in this project is shown in section 4.4.6.

One by one each of these randomly generated genotypes is decoded by the `development` program (see section 4.4) and a network is generated or 'grown' using these decoded parameters. The neural network produced by the network development program is then downloaded onto the MC68000 FORCE board where the network simulator has already been loaded into the MC68000 FORCE board's memory. The simulator is run and evaluated by the network designer. Each network is examined on how the robot responds to its environment and a fitness value (which is an integer in the range 10 to 100) is awarded to the network based on its level of success. The better the robot responds then the closer to 100 the assigned fitness value should be.

When all the networks within the generation have been evaluated, the SGA program is run on the PC to create the next generation of 20 genotypes. To do this the designer is prompted for the fitness value that was assigned to each of the networks. The old population and the fitness values of each of the genotypes are then used to generate the new population using the crossover, mutation and reproduction genetic operators. A detailed description of the operation and the coding of these genetic operators is given in section 4.3.5. In brief however, these operators are at the heart of the operation of the

[5] The description and relevance of each of these parameters is explained in section 4.3.5 which deals with the `development` program. For now it is only important to know that the coding used generates a string of 54 bits. Each bit represents part of a parameter used by the `development` program to grow a neural network . Each bit can assume the value '1' or '0' only.

SGA allowing the production of the next generation based on the performance of the last. This process of testing and evaluation is continued until the algorithm finds a network suitable to control the robot.

4.3.3 Genetic Coding Breakdown

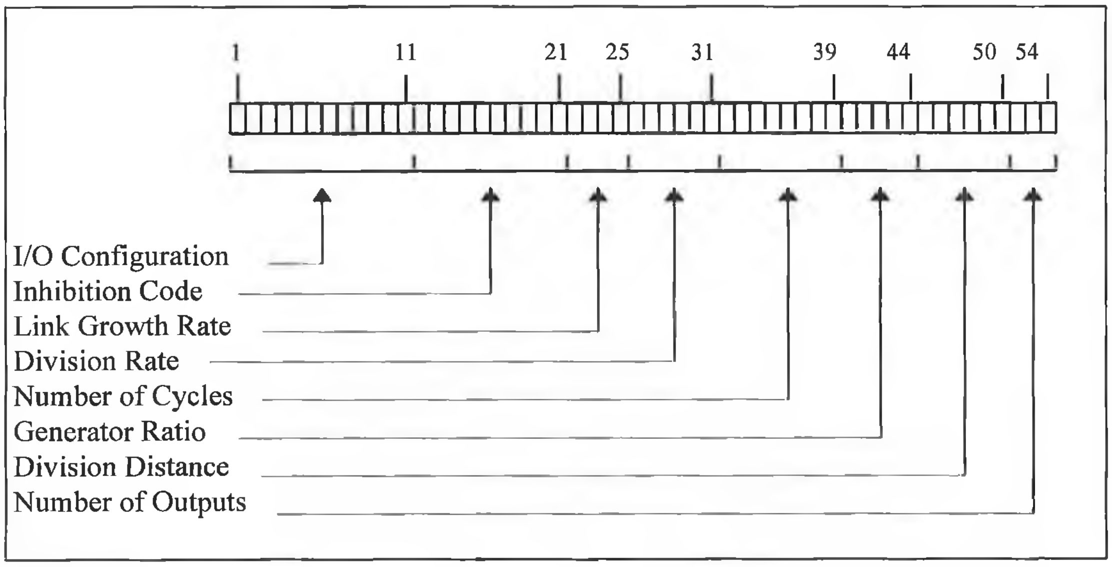


Figure 4.2 Coding of Genotypes Produced by Simple Genetic Algorithm

As mentioned above, the genotypes are 54 bit binary strings. The genotype can be thought of as being a concatenation of a number of smaller binary strings. Each smaller string representing one of the variable parameters used in the development and operation of the neural networks used for controlling the robot. The relevance of each of the parameters is detailed in section 4.4.6. A diagram of the genotype is shown in figure 4.2.

4.3.4 Random Number Generation

Random number generation is used throughout the operation of the SGA. It is used in the initial distribution of the genotypes across the search space. It is also used in the selection of individuals from each generation, based on their performance scores, in order to create the next generation of genotypes. Earlier in the project, the C random number generation routines were used. However, based on advice received regarding the performance of these functions, a different random number generation routine suite was adopted.

The full suite of routines can be seen in Appendix C of this document.

Four of the routines were called from the software used in this project. The routines were called *Get_One_Uniform*, *Get_One_Bernoulli*, *Get_Msr88* and *Set_SeedMsr88*. The code segments for these routines are now shown for completeness.

```
public flag get_one_bernoulli(double p)
{
    int32 cut = MSR88_RANGE;

    if (valid_fraction(p)) cut = fround(p * (double)(MSR88_RANGE));
    else panic("get_one_bernoulli: Invalid parameter <p>!");

    return(get_msr88() < cut);
}
```

Figure 4.2 Get_One_Bernoulli()

```
public int32 get_one_uniform(int32 n)
{
    int32 cutoff = MSR88_RANGE;
    static int32 v;

    if ((n >= 1) && (n <= MSR88_RANGE))
      cutoff = (MSR88_RANGE/n) * n; /* See note in get_uniform(). */
    else panic("get_one_uniform: Invalid parameter <n>!");

    do v = get_msr88();
    while (v >= cutoff); /* See note in get_uniform(). */

    return(v % n);
}
```

Figure 4.3 Get_One_Uniform()

```
public int32 get_msr88(void)
{
  static int32 test; /* static for speed... */
  static ldiv_t lohi;

  count++;
  if (count >= MSR88_RANGE)
  {
    warn("get_msr88: count >= MSR88_RANGE!");
    count = 0;
  }
  lohi = ldiv(seed, Q);
  test = A * lohi.rem - R * lohi.quot;
  if (test > 0) seed = test;
  else seed = test + M;
  return(seed-1);
}
```

Figure 4.4 Get_Msr88()

```
public void set_seed_msr88(int32 newseed)
{
  if ((newseed >= 1) && (newseed <= MSR88_RANGE))
    seed = newseed;
  else warn("msr88_set_seed: invalid <newseed> parameter!");
}
```

Figure 4.5 Set_Msr88()

4.3.5 SGA Operators

It has already been stated that the genetic operators lie at the heart of the operation of the SGA. In this section each of the three operators implemented in this project will be described and its role in the operation of the SGA documented. The three operators used are the reproduction operator, the crossover operator and the mutation operator. Each of these operators uses the random number generation routines described above in section 4.3.4. The implementation of the random number generation will be described, with reference to each of the operators, in this section. The C code implementing these operators will also be shown for completeness.

4.3.5.1 The Genotype Selection Function

Shown below in figure 4.7 is the `select()` function that selects candidates from a population for mating based on the fitness values assigned to individuals. It is based on the `PASCAL` procedure `select` in Goldberg's book[14][p. 63].

```
int *select(double pair_number)
{
        int     j=0;
        int     i=0;
        int     roulette=0;
        double partsum=0;
        int     *mates=NULL;

if ((mates = malloc( 3*sizeof(int))) == NULL)
        {
        printf("Not enough memory to allocate buffer {fn select()} \n");
        exit(1);
            /* TERMINATE PROGRAM IF OUT OF MEMORY */
        }
for(i=0;i<2;i++)
        {
        roulette=(int)(random_floats[i+(pair_number*2)]*
total_genetic_fitness);
                /* VALUE CHOSEN WHICH DETERMINES WHICH CHROMOSOME OF   */
                /* THE OLD POPULATION TO CROSS (OR NOT) WITH ANOTHER   */
                /* OLD CHROMOSOME.                                     */
                /* SEE GOLDBERG FOR DETAILS                            */

        for(j=0;j<MAXPOP_SIZE;j++)
                {
                if((partsum += old_population[j].fitness)> roulette)
                        break;
                if( j == MAXPOP_SIZE-1)
                        break;
                }

        *(mates+i)=j;
        partsum=0;
        }

return(mates);
}
```

Figure 4.6The SGA Select() Function

The `select()` function uses a pre-constructed array (random_floats) to select a value (X) which lies between 0 and the sum of the assigned fitness values of all the individuals in the population. The function then iterates through the population index, adding each individual's assigned fitness value to a temporary sum figure(Y) which is initially zero. When the temporary sum figure(Y) exceeds the randomly generated number(X) then the individual responsible for the last addition to (Y) is selected for reproduction. What this means is that the higher an individual's fitness figure is, then the higher the chance that the addition of that particular individual's fitness figure will result in the randomly generated number(X) being exceeded by the temporary sum(Y), and hence be an individual picked for reproduction. This operation is described by Goldberg using a roulette wheel analogy[14] which intuitively may be easier to understand than this purely textual description.

The reproduction operator is an artificial version of natural selection, a Darwinian survival of the fittest among string creatures. In natural populations, an individual's fitness figure is determined by its ability to avoid disease, predators and other obstacles to adulthood and subsequently mate. In this artificial setting the fitness figure is the sole arbiter for the survival of the string creatures.

This function is called twice within the user defined function `crossover()` (see 4.3.5.3) to select a pair of individuals for breeding. When two distinct individuals have been selected then the crossover between the genotypes is carried out within the `crossover()` function.

4.3.5.2 The Mutation Function

Shown in figure 4.8 below is the code which implements the mutation function. This function is also based on the equivalent Pascal procedure in Goldberg[14][p. 65]. Within a simple genetic algorithm implementation the mutation and crossover operators can be used either mutually or exclusively. For example, Dawkins' implementation in the creation of his biomorphs[9] does not use the crossover operator at all. Instead he depends purely on a mutation operator and a reproduction operator. In this implementation, both are used.

```
int mutation(int single_gene)
{
      int new_gene;      int mutate;

mutate = flip(PROB_MUTATATION);
            /* FLIP SIMULATES A WEIGHTED COIN TOSS AND IS USED    */
            /* HERE TO DETERMINE WHETHER A SINGLE BIT SHOULD BE   */
            /* MUTATED OR NOT. FLIP USES THE GET_ONE_BERNOULLI()  */
            /* FUNCTION                                           */
if (mutate)
      {
      textcolor(RED);
      if (single_gene == 1)
            new_gene = 0;
      else
            new_gene = 1;
      }
            /* IF THE FLIP FUNCTION RETURNS A '1' THEN THE BIT IN */
            /* QUESTION IN THE CHROMOSOME IS INVERTED             */
else
      new_gene = single_gene;

#if DEBUG
      mutate_number++;
#endif

return(new_gene);
            /* RETURN THE VALUE OF THE NEW BIT                    */
}
```

Figure 4.7 Implementation of the SGA Mutation operator

The mutation operator in this implementation is present purely to prevent important information being lost. This can happen as the SGA population begins to approach an optimum solution in the search space. As the optimum is approached, all the genotypes begin to resemble each other. This is fine, as that is what the SGA is supposed to do. However, if one is dealing with a search space which has more than a single optimum, the SGA could be converging on a local optimum rather than the search space optimum. This concept of multiple optimums is explored in more detail in Goldberg[14]. What the mutation operator does is to change, at random, small parts of individual genotypes. If used correctly[6], the operator can act as an insurance policy for the SGA, potentially preventing it from converging on a local optimum, rather than the search space optimum.

The coding implementation of the operator is simple once the mutation rate for the population has been chosen (see section 4.3.5.4). In the crossover function, each bit of each genotype is passed into the function `mutation()`. The random number function `flip()` is then used to decide whether or not the bit is to be mutated. The probability of mutation is passed as a parameter `to flip()`, as it is also used to select a crossing site for parents in the crossover function.

4.3.5.3 The Crossover Function

The `crossover()` function (see figure 4.9) implements the crossover operator. It uses the results of the `select()` function as two of the input parameters. It also uses the `mutation()` function internally to implement this form of crossover. It is based on the equivalent PASCAL procedure in Goldberg[14][p. 64]. A thing to note is that the function does not allow generations to overlap. This means that genotypes from the old population (from whose parts the new generation is constructed) may not cross with the

[6] The choice of genetic operators can be biased within any particular application to whichever the designer chooses. For example in Dawkins' work, the mutation operator is used to the exclusion of the reproduction genetic operator [9].

new generation's genotypes during the creation of the new population of individuals. The reasoning behind this choice was to make the implementation of the SGA as simple as possible. As was mentioned before, with reference to the biasing of genetic operator influence, the crossover function can also be made much more complex. It can, for example, involve cross generation coupling as well as self reproducing genotypes if required.

```
void crossover(int mate1,int mate2,int crosspoint)
{       int cross;    int xsite;    int iter;
        int parent1[GENE_LENGTH];  int parent2[GENE_LENGTH];

for(iter=0;iter<GENE_LENGTH;iter++)
        {
        parent1[iter] = old_population[mate1].chromosome[iter];
        parent2[iter] = old_population[mate2].chromosome[iter];
        }
                /* GET THE TWO CHROMOSOMES CHOSEN FOR REPRODUCTION OR*/
                /* CROSSINGFROM THE OLD POPULATION RECORD              */

cross = flip(PROB_CROSSOVER);
                /* FLIP SIMULATES A WEIGHTED 'COIN TOSS' AND IS USED */
                /* HERE TO DETERMINE WHETHER THE TWO CHOSEN          */
                /* CHROMOSOMES SHOULD BE CROSSED TOGETHER OR NOT     */

if (cross)     {xsite =  crosspoint;}
else
        xsite = GENE_LENGTH;
                /* IF THE CHROMOSOMES ARE NOT TO BE CROSSED THEN THE */
                /* CROSSING SITE CHOSEN IS SIMPLY THE END OF THE TWO */
                /* CHROMOSOMES.                                      */

gotoxy(xsite+7,(popsize*2) + 4);
printf("x");  /* PRINT POSITION OF CROSSING SITE ON SCREEN           */

for (iter=0;iter<xsite;iter++)
    {
    textcolor(GREEN);
    new_population[popsize].chromosome[iter] = mutation(parent1[iter]);
    gotoxy(iter+7,(popsize)+3);
    cprintf("%d",new_population[popsize].chromosome[iter]);
    textcolor(WHITE);
    new_population[popsize+1].chromosome[iter] = mutation(parent2[iter]);
    gotoxy(iter+7,(popsize)+4);
    cprintf("%d",new_population[popsize+1].chromosome[iter]);
    }
                /* THIS LOOP GENERATES TWO NEW MEMBERS OF THE        */
                /* POPULATION AND AS EACH BIT OF THE NEW CHROMOSOMES */
                /* IS GENERATED BY THE CROSSING IT IS DETERMINED     */
                /* WHETHER OR NOT IT SHOULD BE MUTATED               */

if(xsite!= GENE_LENGTH)
        {for(iter=xsite;iter<GENE_LENGTH;iter++)
                {
                textcolor(WHITE);
        new_population[popsize].chromosome[iter]=mutation(parent2[iter]);
                gotoxy(iter+7,(popsize)+3);
                cprintf("%d",new_population[popsize].chromosome[iter]);
                textcolor(GREEN);

        new_population[popsize+1].chromosome[iter]=mutation(parent1[iter]);
                gotoxy(iter+7,(popsize)+4);
                cprintf("%d",new_population[popsize+1].chromosome[iter]);
                gotoxy(5,popsize+3);
                }
        }
                /* IF THE CHROMOSOMES HAVE BEEN CHOSEN FOR CROSSING        */
                /* THEN THIS WILL IMPLEMENT THE ACTUAL EXCHANGE OF         */
                /* BITS. IF NOT, THEN THE PROGRAM WILL NOT ENTER THIS LOOP. */
}
```

Figure 4.8Function in C to implement the crossover operator

4.3.5.4 Genetic Parameter Choices

So what values were chosen for the probabilities of crossover and mutation occurring? To choose these numbers, the work of Goldberg was again addressed. Goldberg references a study by De Jong [12] in the application of genetic algorithms to function optimisation. He states that De Jong recommends the choice of a high crossover probability coupled with a small mutation probability (inversely proportional to the size of the population) and a moderate population size [14]. Following these suggestions, the following figures were chosen:

Probability of Mutation:	0.01
Probability of Crossover:	0.9
Population size:	20

Figure 4.10 Genetic Parameters Used

These figures were chosen to allow the SGA to converge quickly (even to a local optimum) to demonstrate the viability of the overall design approach.

4.3.6 Summary

The Simple Genetic Algorithm was an adequate choice for the project. It is entirely possible of course, that there is a different form of the genetic algorithm that would be even more suited to artificial neural network design. However, as has already been explained, only the briefest examinations could be made of each of the theoretical areas employed due to the time scale of the project. The implementation of the SGA was aided by the conscious decision to avoid over-complication of the core code. The operation of the code itself was validated early in the project[25]. The working of the final code in optimising the operation of the neural networks is discussed in detail in section 4.5 and thus will not be addressed here. However, it would be appropriate to say at this stage that the operation of the SGA code was a success with respect to the results obtained.

4.4 The Network Development Program.

4.4.1 Introduction

After the manipulations of the simple genetic algorithm, the network development program decodes the genotypes created by the SGA. It uses the decoded genotype parameters to create or 'grow' artificial neural networks. This growth process is deterministic. It is based on some very simple rules that use the growth parameters encapsulated in the genotypes.

This section details how these network growth parameters are decoded from the genotypes. It also describes the rules governing the creation of a neural network based on these parameters.

The inspiration for this approach is the work of Richard Dawkins [9,10,11], whose work is in this general area, is also outlined in section 2.5.

4.4.2 Origin of the Idea.

As stated in the introduction of section 4.4, the idea of growing a neural network is based on the work of Richard Dawkins [9,10,11]. Dawkins' work in this project has been emulated but instead of re-creating Dawkins' biomorphs[7], neural networks were to be grown. To accomplish this a set of rules had to be constructed governing how the neural networks were to be grown from the genotypes. These rules were constructed in such a way as to attempt to make the growth of the neural networks somewhat biologically inspired. This of course, is not the only way in which the development of the neural networks could be done. In Sussex University a research team have implemented a low level behaviourally based robot. However, instead of using a growth mechanism, such as in this project, they coded every connection in the neural network into the genotype. Further details of their work can be found in [6,7,8,15,16,17,18,20].

As well as defining the rules for growth, it was also necessary to decide what parameters were to be encoded in the genotypes in order to facilitate this biological approach.. These parameters would be used by the rules to grow the networks. The rules used are listed in section 4.4.3. These rules have been used successfully to generate a neural network for robotic control. However, as with all areas of this project, simplifications were made in order to precipitate the implementation success. These simplifications generated some problems and these problems will be discussed in this section where it is relevant to do so.

[7] Biomorphs are what Dawkins calls his biologically inspired pictures.

4.4.3 How Does It Work? (The Rules).

As stated above, a number of rules were devised in order to make the growth of a neural network from a genotype possible. These rules resulted in the growth of the neural network being deterministic and hence repeatable. It was a conscious decision to make the growth a deterministic process as repetition of the results obtained was very important at this early stage. It was important to qualify any results obtained to validate the procedure being implemented in the project. The rules will now be presented. The rules have been broken down into three groups to make them more readable. The first group describes the general rules governing the overall operation of the growth process. The second and third groups deal with the growth of links between nodes and the generation of the nodes themselves respectively.

4.4.3.1 General Rules

1. The neural networks are made up of two parts: nodes and links.
2. The nodes are of the types described in section 4.5.5.
3. Links grow to form connections between nodes.
4. The networks are grown on a growth 'grid' of size 255 x 255 units.
5. Each node occupies 1 unit on the grid and remains set in that position.
6. Each node can only attempt to create a connection with one other node at a time (i.e. only has one link growing at a time).
7. The position of each node and link is stored as a set of (x,y) co-ordinates on the grid.
8. Six nodes exist initially on the grid in fixed locations, though not in fixed order. Four of these are input nodes that represent the physical connections to the robot's

sensors and two are output nodes that represent the connections to the robot's motors. Which node corresponds to which sensor or motor is determined by a decoding of the genotype.

4.4.3.2 Link Growth Dynamics

1. The current link (only one link grows at a time) of a node grows towards the nearest node relative to the tip of the current link that is not the node from which the link is growing - referred to as the parent node.
2. On each growth cycle the location of the nearest node is re-evaluated. This re-evaluation is to allow for the placement of new nodes due to node division.
3. The link growth rate is constant for all links and is defined by the genotype.
4. Nodes can only connect (via the links) with a range of 1 to 10 other nodes as defined by the 'number of outputs' growth parameter. (See figure 4.10).
5. When links do make a connection between nodes, the distance between the base node and the newly connected node is the magnitude of the weight of the connection between the nodes as used in the simulator (maximum value 255).
6. The sign of the connection weights is determined by decoding the genotype to determine if the connection is an inhibitory link or not. This is defined by the inhibition code parameter which is described in section 4.4.6.

4.4.3.3 Node Division Dynamics

1. Node division takes place at a constant rate to produce new nodes. This rate is defined by the Division Rate parameter. See section 4.4.6.

2. Each node can divide once to produce only a single offspring, and the parent node may continue to grow links after divisions up to the allowed maximum.

The positioning of newly created nodes, on division from the parent node, is determined by the following rules:

1. The division distance is constant for all node divisions and is decoded from the genotype. This is defined by the Division distance parameter.
2. The division direction is determined by finding the nearest two nodes relative to the parent node and finding the point that lies half way between them. See figure 4.11 for an illustration.
3. The position at which the new node is placed is at the division distance away from the parent node in the direction of the point derived above.
4. If the position of the new node lies outside the growth grid then the new node is placed within the growth grid. This is accomplished by determining which of the new node's co-ordinates has violated the boundaries of the growth grid. The offending co-ordinate is then adjusted so that it is brought back within the boundaries of the growth grid.
5. If the position calculated for the new node is within a radius of 30 units of another node then the division is deemed to be invalid and the new node is not created. This is to prevent 'clustering' of nodes in small areas of the growth grid. It was decided that clustering could have prevented connections being made between the input and output nodes. It was decided that this was best avoided at this early stage of overall framework development.

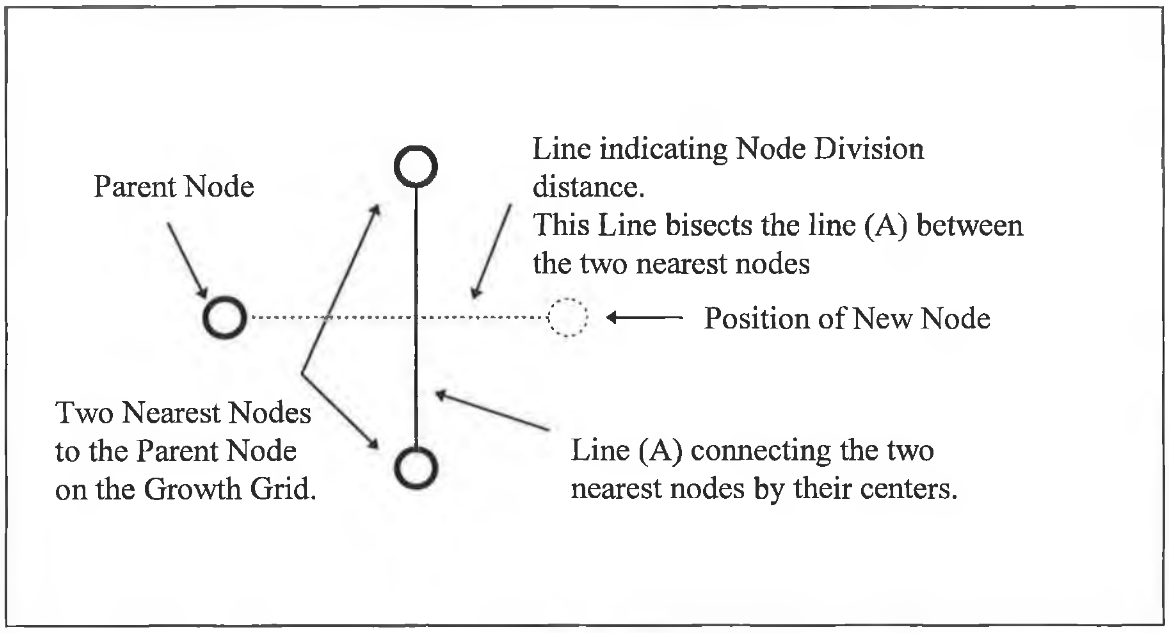

Line indicating Node Division distance.
This Line bisects the line (A) between the two nearest nodes
Parent Node
Position of New Node
Two Nearest Nodes to the Parent Node on the Growth Grid.
Line (A) connecting the two nearest nodes by their centers.

Figure 4.9 Illustration of node division dynamics.

4.4.4 Implementation of the Rules

Implementation of these rules within the development program produces 3 different sets of data (arrays) as a final result. The arrays are:

1. An array to record the position of every node on the grid.
2. An array to record all connections between nodes.
3. An array to record the connection weight between every pair of connected nodes.

These three arrays are then combined and the format of the combined array is adjusted. The new format is adjusted such that the single final array contains all the information in the format required by the neural network simulator and simulation[8] software. This is the array which is downloaded onto the MC68000 SBC and then tested on its ability to control the robot. See chapter 5 for the behavioural evaluation rules. It takes a slightly

[8] An explanation of the differences between the neural network simulator and simulation software, and the uses of the simulation software in the overall framework is given in section 4.5.

different form when being used in the simulation software. An number of examples of how the grown networks 'looked' are given in Appendix E. The download of the single formatted array will now be described.

4.4.5 Network Download

The download of the network to the MC68000 SBC is a simple operation. The PARAGON C cross compiler[28] used to generate the simulator code also generates a memory map listing the location of all functions and global variables/data structures. When the simulator code and the SBC are working correctly then re-loading of the simulator software is unnecessary within each generation evaluation run. This is because the memory locations of the variables, data structures and functions remain fixed. Hence every network grown by the development program can be downloaded and slotted into the same memory location to be used by the simulator already loaded in the MC68000 SBC's memory.

To allow the network download to the MC68000 SBC to be performed it was necessary to write a C program which generated a file containing the network information in S19 format [28]. This is the format required by the MC68000 SBC's operating system to allow recognition of the incoming data. Use of the S19 specification requires the development program to perform conversion of the integer figures produced by the development program to hexadecimal character format and the calculation of checksums and address offsets. The checksum calculated allows the MC68000 SBC to check if download data integrity has been maintained. The offset generated is used to inform the

MC68000 system of where to store the information in memory so that the simulator can access it correctly.

Reference was made earlier in this section to the neural network simulation software. To use the information from the development of the network in the simulation program a slightly different approach is taken. The PC on which the simulation is run does not require S19 format downloads. The development program grows the network in the same way and reduces the three arrays to the format required by the simulation and simulator software. Then it outputs this array as a text file before the conversion to S19 format is carried out. This text file then contains the array information in the format that the simulation software requires. The array is read directly from this file by the simulation program.

To allow this difference in information transferal between the simulation and simulator software, the development program has two separate versions. They are each contained on the disk accompanying this report. The programs are called PCDEV.EXE which produces the networks for the simulation software and DEVEL.EXE which produces the S19 format file (through use of the FORMS19.C program) for download to the simulator on the SBC.

In the next section the mechanism for the decoding of the genotypes produced by the SGA to grow the networks will be described.

4.4.6 The Genotype Parameter Decoding.

The SGA works on a coding of a parameter set, not the parameters themselves. This section deals with how these genotype parameters are decoded and what they represent to the development program.

The decoding of some of these parameters is simple and the decoding of some is not so straightforward. The creation of the final format in which the parameters have been coded in the genotype was a gradual process. A number of decisions had to be made with regard to what exactly should be coded into the genotype. The most difficult area to decide this in was with regard to the parameterisation of the individual nodes within the neural network. The neural nodes have a number of internal parameters which directly affect their performance. It was decided to minimise the parameterisation of the nodes in order to reduce, as much as possible, the size of the genetic search space. It was hoped that this reduction in variability would facilitate a more speedy convergence to some optimum on the search space. This convergence would then, hopefully, validate the operation of the design framework. The full details and reasoning behind this reduction in variability will be given in section 4.5.4.

Shown below is figure 4.12 which has been reproduced for convenience to show the location and name of the different parameters that are coded within each genotype produced by the genetic algorithm.

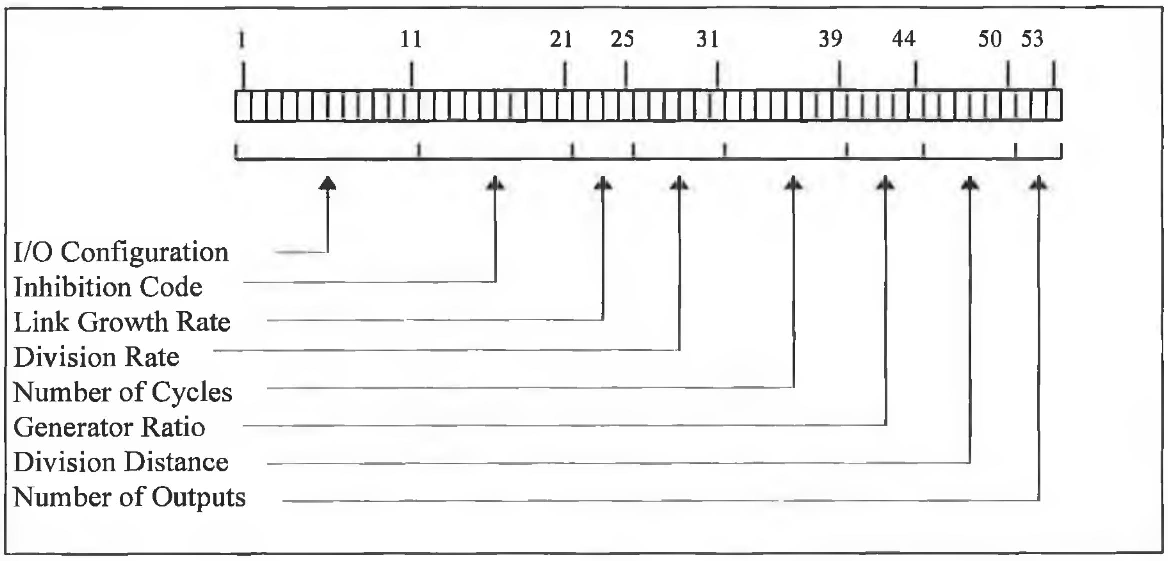


Figure 4.12 Coding of Genotypes Produced by Simple Genetic Algorithm

How the individual sections are decoded from the genotype is now listed parameter by parameter

1. *I/O configuration:* Bits [1-11]. This parameter determines which sensor is placed in which position on the growth grid. The algorithm can be observed in the function `int *get_io_posit(int)` in the GET_DATA.C program. The algorithm is quite simple in nature. It uses the bits 7-11 to select the first node to be placed on the grid(1-6). The remaining bits determine the order in which the remaining nodes will be placed on the grid.
2. *Inhibit Code:* Ten bits [12-21] which determine which links are to be positive and which are to be negative. A maximum of ten connections is allowed so one bit corresponds to each link. If the bit is a 1 then the weight is positive. If the bit is a 0 then the weight is negative.

3. *Link growth rate:* The bits [22-25] are translated into an integer between 1 and 16. This is the distance which links grow in a single iteration of the development growth routine. It is possible that with a growth distance of 1 that the link head position will be the same after calculation as the current position of the link head. This is caused by rounding errors. To overcome this, the result is analysed and if the error is detected the growth distance is increased to 2 units and recalculation takes place.
4. *Node division rate:* The five bits [26-31] are converted into a decimal integer between 1 and 16. This number is multiplied by 10 to give a range of 15 numbers between 10 & 150. This is the number of iterations of link growth that occur between node divisions.
5. *Number of Growth Cycles:* The eight bits [32-39] are translated into a decimal integer between 1 and 256.
6. *Generator Ratio:* The 5 bits [40-44] are used to determine which nodes act as generator nodes[9] . The ratio can be between one in every 2 to 33 nodes.
7. *Node division distance:* The four bits [45-49] are first translated into a decimal integer between 0 and 15. This number is then multiplied by 4.375[10] and 30 is added to the result. This means that the division distances can take one of sixteen numbers between 30 and 100 units.
8. *Number of Outputs:* The four bits [50-53] are used to indicate how many outputs which each node can have. Ranges from 1 to 16.

[9] See section 4.5.5.2 for description of the operation of generator type nodes.

[10] This factor converts the binary number distribution from the range 0-15 to 0-70.

4.4.7 Network Growth.

The main function that controls the growth of the neural network using the decoded parameters is shown here as it is the most concise way of describing its operation.

```
void sim_growth(int ax_grt,int div_rt,int div_ds,int *inbt,int
num_cycles)
{
      int count=0;
      int iter=0;
      char type;
      char msg[20];

while (iter < num_cycles)
      {
            /* pass in link growth rate and inhibit information */
      grow_links(ax_grt,inbt);
      if (count>=div_rt)
            /* if time for node division then begin */
            {
                  /* limit on number of nodes allowed */
            if (num_nodes>(MAX_NODES-6))
                  stall();
                  /* function to exit program gracefully */
            else
                  grow_nodes(div_ds);
                  count = 0;
            }
      count++;
      iter++;
      }
}
```

Figure 4.10 Controlling function for Network Growth.

Some of the basic graphics functions that are included in the C libraries available to the Borland and Turbo C++ compilers were used in the development program to allow visualisation of the program's operation during debugging. This graphics element is maintained within the program as it shows the person using the program that something is happening while the network is being grown. This, I feel, is preferable to leaving the screen blank and giving the user no indication that a network is or is not being grown successfully.

4.4.8 Summary

So why were these particular rules used, and how has the development program performed? A number of decisions needed to be made regarding this particular question as the project progressed. Some of the potential decisions were raised in 4.4.2 and in this section these proposals will be reviewed and others that presented themselves at a later stage addressed also.

One of the first decisions was in reference to the node division mechanism. It had previously been thought that the use of the two nearest nodes relative to the node about to divide to decide the direction in which the new node should be placed may not have been good. It was thought that this method could lead to a situation where if the node division distance was small then when the time comes for the new node to divide, it will attempt to place its child node in the exact position of its own parent. This, it was thought, would minimise the number of nodes which could occupy the growth grid even when a lot of grid space remained unfilled. The maximum number of nodes that can occupy the growth grid (assuming a minimum radius of 30 units between all nodes) is 64. In the testing done to maximise the number of nodes in a grid the maximum that could be reached using the SGA was 25. On first glance this would seem to confirm the worries highlighted earlier. It was thought that a change to select perhaps the 2nd and 3rd nearest rather than the 1st and 2nd nearest to derive the node division direction may have unleashed more of the growth potential. However, it was decided to trust in the operation of the SGA. Indeed, as the results appear to confirm (see chapter 6) this was a vindicated choice.

It was also thought that it may not have been good to have the location of the initial nodes static on the grid. It was decided to allow this variability in the genotype coding and to use the SGA to 'find' the best location for the opening configuration. The results on this point were a little more difficult to evaluate. This point is discussed in more detail in the conclusion.

Another parameter which made its appearance late into the project was the introduction of the generator nodes ratio. It was decided to use generator nodes to enable the system to generate neural networks which would not require an external stimulus in order to begin to operate. It was decided instead that some form of spontaneous operation would be preferable to kick-starting every network. This was because the entire operation and biasing of the network could have been determined by which external sensor was stimulated first.

4.5 Simulator and Simulation

4.5.1 Introduction

In this section the operation of the neural network simulator and simulation software will be documented. The simulat*or* software was used to implement the artificial neural networks that were created. It was designed to operate on the SBC controlling the robot. The simulat*ion* software was used to debug the simulator software by emulating the operation of the SBC on the PC. This allowed the neural network software to be evaluated without the framework hardware considerations that have already been documented in this thesis (see chapter 3) influencing or impeding its development. A number of difficult situations arose over the period of development of the simulator software. Some of these problems have already been referred to in previous sections - the genotype coding and operation of the SGA affecting the level of parameterisation possible for the neural nodes for example. Difficulties caused by the differences in compilers (Borland C and Paragon C Cross Compilers) occurred. Other difficulties arose prompting decisions about the operation of the network based on initial results. All these difficulties affected the overall development of the software from its original aspirations up to its final state. In this section of chapter 4 the more difficult problems encountered and their eventual resolutions will be documented.

Also in this section, the operation of the neural networks will be described in some detail. Reference will be made to the work of Randall Beer[2] where appropriate. This is because it was intended to use the type of networks modeled by Beer to control the robot. Beer's network model was the ideal originally, but as the problems mentioned above revealed themselves, it became necessary to make some quite significant changes to the network morphology and operation. As much as possible however, Beer's model was adhered to. The eventual mechanics of the connections allowed between nodes and the various types of nodes possible at the end of the research period will be described. As the operation of the networks is integral to the success of the project, the operation of the networks will be described first. Comparisons to Beer's model will then be made.

4.5.2 Operation of the Neural Network.

So how does the neural network operate? As stated above, the neural network operation was modeled on the network types designed by Randall Beer. The network, as with most artificial neural networks, uses a simple input/output interface mechanism. Between the inputs and the outputs lie a number of neural nodes of various types. The various types of nodes implemented will be described later in section 4.5.5. The nodes are connected to the input and output sections of the neural network and also to each other by weighted links. These links model the axons that form the connections between neural cells in real neural circuits. There is no defined structure to the neural network in the sense that may be understood for a typical feedforward neural network for example[29,31]. Figure 4.14 below shows the general structure of such a network.

As can be seen from figure 4.14, the more traditional feed forward artificial neural networks have a strongly defined morphology. There are no feedback loops or any deviation from the connectivity structure. Certainly, one or many of the connections could potentially have a connection weight of 0, which would mean that the connection was in essence not present. However, that does not detract from the true design structure of the overall network. The nodes themselves are also quite simple in this type of network. One of the more common internal nodal dynamics that is implemented in such networks is a simple scaling function. This scaling can be a simple as a normalisation of the input received by a node to an output range of 0 to 1.

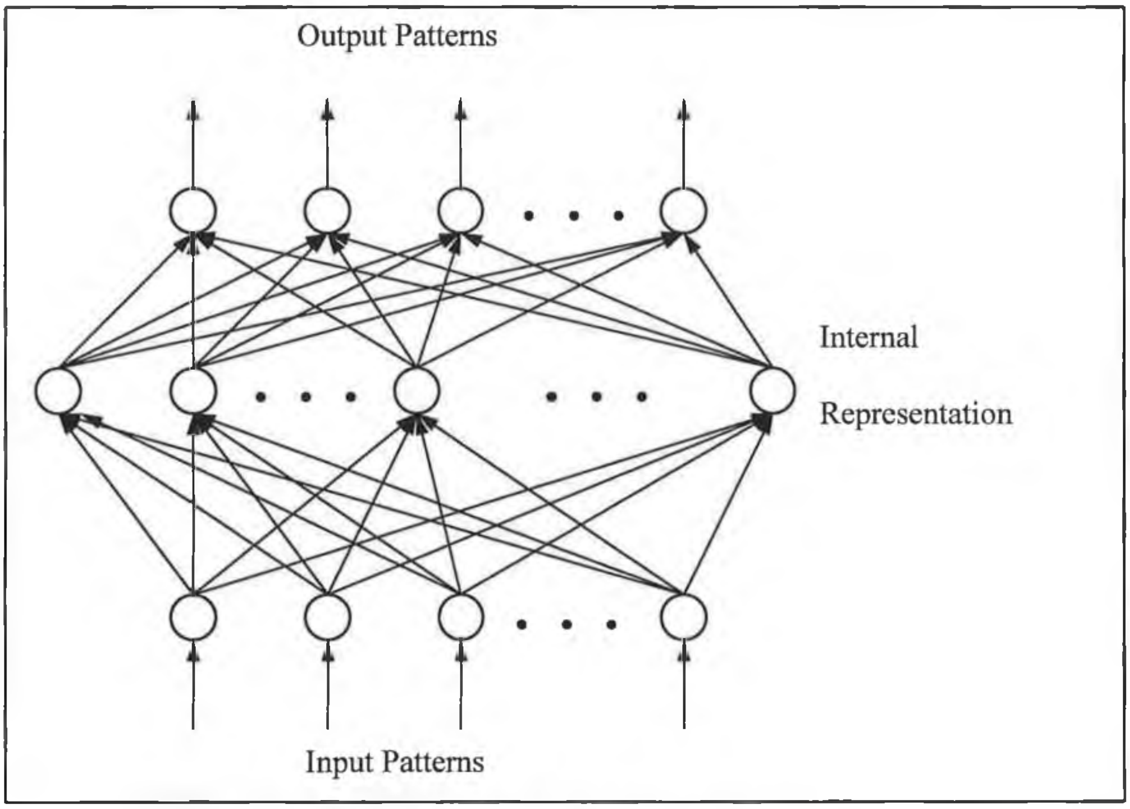


Figure 4.11 Typical Feedforward Network Architecture.

The networks designed by Beer and modeled in this project are of a radically different structure. The morphology of these networks is not as regular as the morphology of the type of networks exemplified by the traditional feedforward networks described above. The input nodes and output nodes are obviously similar in that they define the inputs and outputs delivered to and produced by the network operation. However, the morphology of the internal network 'layers' within these networks is much more varied. The architectures employed in artificial neural networks tend be fairly homogenous.

That is, they consist of a number of formal neurons connected in some uniform way. In contrast, in real neural networks, connections between nodes tend to be very specific and highly non-uniform. To illustrate, it is possible to have feedback loops within the internal ‘layers’ of the networks. It is also possible to have connections between any two nodes in the network, making it conceivable to have connections from the inputs to the outputs of the network directly.

Furthermore, in real neural networks, individual nerve cells have often unique response and internal properties. Their response is influenced by the morphology of their connections, by the types of channels between them and other nodes and the electrical and physical properties of their cellular construction. Beer states that there is considerable evidence that that both the individual cellular properties and their specific interconnectivity are crucial to the ways in which individual neural circuit's function. He references Selverston[12] and Llinas[22] to support this claim.

As mentioned in the introduction to this section, it was envisaged that the design of the neural networks used would be closely based on the networks implemented by Beer. Beer’s networks demonstrated successfully some of the simple behaviours displayed by real insects. He produced networks that controlled his simulated insects in ways that were similar to the edge-following behaviour displayed by real insects. Other networks allowed the simulated insect to track down ‘food’ in its artificial environment. Indeed, Beer mentions the work of Brooks as a possible avenue for the expansion of his research into this area of artificial behaviour. However, the successes displayed by Beer’s work had some considerable limitations in their application to real time robotic control. The networks designed by Beer operated in an artificial environment and the time scale in which its simulated insects behaved was, according to Beer himself, almost ten times slower than would be expected in real time[2]. This brought into question the feasibility of their use at all in such an application as real time robotic control.

It was decided that it would be interesting to pursue their use in such an application however. The only obvious way to do this, given the limitations on available computational power, was to simplify the network dynamics so that the networks could be updated in real time. Also, Beer's simulation was concerned with illustrating graphically the movement and behaviours of the artificial insects. It was hoped that the absence of the graphics that were involved in Beers simulation would mean that it would be possible, potentially, to implement the network model with very little change. Obviously, with no quantitative figures for the time costs of the graphics used by Beer, this decision would have to be reviewed in the light of experience. The end results however, appear to vindicate the decision. So what simplifications were introduced?

One of the most complex, and computationally expensive, areas of network operation is in the implementation of the individual nodal dynamics. The operation and update of the interconnections seemed computationally cheap in comparison. Beer himself describes their complexity as lying about half way between the types of model neurons used in computational neuroscience and the more traditional model neurons used in the likes of feedforward networks. This seemed the ideal place in which time could be saved. Therefore, it was decided to sacrifice some of the internal properties of the model neurons implemented by Beer in order to facilitate a real time implementation.

However, before this simplification and the others that followed it are detailed, the development of the simulator and simulation software will be described.

4.5.3 Simulator and Simulation Development

At the early stages of the simulator development, networks were designed by hand using the networks designed by Beer for examples. These networks allowed the simulation and simulator software to be tested. Implementation of any of these networks, designed either by hand or by the use of simple genetic algorithms, was through the use of the simulator or the simulation software. These two programs share almost completely the same internal functions.

As stated previously, the differentiation between the two programs is that the simulat*or* software was written to run on the Motorola MC68000 Force SBC used to control the robot and the simulat*ion* runs on the PC. Phrased most simply; the simulat*ion* software models, on the PC, the input/output behaviour of the simulat*or* on the SBC. The simulation software was written to allow the operation of the neural networks to be visualised quickly during the debugging of the routines common to both. This would exclude the time consuming task of cross-compilation and information downloads to the SBC every time a change was made. Also the simulation allowed the neural networks designed by hand to be tested during their design stage without having to generate S19 format files. This allowed them to be used later in the project to test the overall operation of the hardware framework.

One of the major difficulties that arose in the past was due to the different compilers used for the two programs. The compilers used are the 'Turbo C++' compiler for the simulation and the 'PARAGON C-cross compiler' for the simulator running on the SBC. The simulator and simulation software appeared to be operating differently despite the same code being used in both pieces of software. Eventually, the problem was tracked down to the data type conversion routines used by each compiler. It was realised that great care must be taken in cross data type operations, as different compilers have different conversion routines to deal with these situations. For example, when multiplying a real number by an integer, is the integer converted to a real number type first, or is the real number truncated to form an integer? This problem was avoided by defining the types in the conversion explicitly using casts[11].

Prompted by the discrepancies due to type differences, and the reduction in network parameter variability required to allow use of the SGA search technique, the simulator and simulation software underwent a number of structural changes over the course of the project. The final versions of the simulator and simulation software are on the disk accompanying this thesis. The simulator code is in the file NETLESS.C and the simulation code is in the file PCSIM.C. The simplifications that were carried out on the neural network model will now be described.

[11] A cast is an operator used to convert the data type of a variable explicitly. E.g. `(int) (x)` converts the variable x to a type 'integer' in the C language.

4.5.4 Simplifications

To recap before continuing, it was stated in section 2.3.2 that the neuronal model used in this project was based on the neuronal model used by Beer. It was also mentioned that certain simplifications to the nodal model and the network morphology became necessary. They was necessary in order to allow any chance of a real time implementation of robotic control. They were also required in order to reduce the size of genetic search space in which the genetic algorithm had to operate. For these reasons a number of simplifications took place over the course of the project. These simplifications will now be detailed. A pictorial comparison between Beer's model and the model implemented in this project is shown in figures 4.19 and 4.20.

4.5.4.1 Simplification 1 (Internal Conductances)

The first simplification made was the removal from the neural node structure of the internal conductance parameters. It was felt that their potential effect on the network operation could be disregarded. These internal conductances apparently affect the time dependent input response characteristics of real neuronal cells as well as spontaneous activity. Beer cites the work of Selverston[32] to illustrate this. Seleverston states that these properties appear to be crucial to the function of those neural circuits that have been analysed at a cellular level. However, it was felt that the computational cost of their inclusion in the network model for this application would have outweighed their usefulness at this early stage of development. It was decided to rely solely on the other properties of the nodal model. Later in the development of the software, generator nodes were introduced to allow spontaneous behaviour to occur in the network. This type of node is described in section 4.5.5.2.

4.5.4.2 Simplification 2 (Integers).

The very first revision that was carried out on the network operation was in the conversion of all possible variable types from floating point types to integers. This revision to the simulator and simulation software was for two reasons. The first reason was the problems caused by the type conversion differences that exist between the different compilers used for the simulator and the simulation. By making all the variables the same type, these problems could be avoided. The second reason for the conversion was that integer operations are carried out much faster on the SBC due mainly to the low number of bits used by the SBC microprocessor to represent them combined with the built in microprocessor hardware used to process them. This is advantageous in a time conscious project such as robot control.

Due to this conversion to integer type, some of the original neural network parameter specifications had to be adjusted. The specification for network weights remained at ±255 (integers only), but obviously the neural nodes could not continue to output a value between 0 and 1 if integers were the only type to be used in the software. The nodal output range was thus adjusted to assume a value between 0 and 255 (integers only). The increased magnitude of the output value range meant that the threshold parameter (which previously took a value between 0 and 255) would now take values between 0 and 65,035. The minimum output level parameter (see section 4.4.7) was affected in a similar way and thus had its range changed from 0 to 1 to the range 0 to 255.

A neural network was designed using these new parameters. To do this, a previously designed and operational neural network was converted. The design of this new neural network was necessary to ensure that the simulator (and simulation) software was still capable of implementing a neural network to control the robot. The design took time (since it was done by hand) but was completed successfully verifying the changes made to the programs.

Regarding the attempt to decrease the time required to update the network, no discernible difference could be detected in the operational speed of either program. It must be noted however that the networks being designed at that point were small and took very little time to update anyway. Therefore it is not surprising that speed differences were undetectable to the human eye after the change. No benchmarking was done at this point to quantify the speed increases. It was deemed more necessary to press on with the overall goal of the project than to continually stop and analyse in detail what had been done previously. This almost certainly led, in cases such as this for example, to spending valuable time on perhaps trivial enhancements to the network update performance. It was felt however, that this was an acceptable sacrifice in light of the overall goal of the project.

4.5.4.3 Simplification 3 (Neural Node Structure).

The decision to use simple genetic algorithms and a development program, instead of hand designing networks, meant that a second revision was required. This time the revision was focused on the node used to model neurons within the simulator and simulation programs.

It has already been stated that it was decided to model as closely as possible the model used by Beer. The neural node used in the simulator and simulation has the same non-linear input/output gain characteristic as Beer’s model (see figure 4.15). The routine used to update the input figure[12] for the neural nodes was simplified. The input figure in Beer's work was updated using a differential equation[2][pg. 51]. This requires numerical analysis techniques to be solved. The simplification of the node modeled in this project resulted in this update routine being represented by a difference equation. This meant that the update was made numerically simpler and, more importantly, faster. This modification was required to allow the network to be updated in real time on the robot, although it also detracted from the biological plausibility of the nodal model.

[12] The input figure for a node models the charge which the membrane of a real neuron stores. The membrane of a neuron behaves like an RC circuit; the charge builds up if a node is constantly receiving inputs, and when the inputs stop the charge decays. Beer uses a RC circuit to model this behaviour [2][pg 50]·

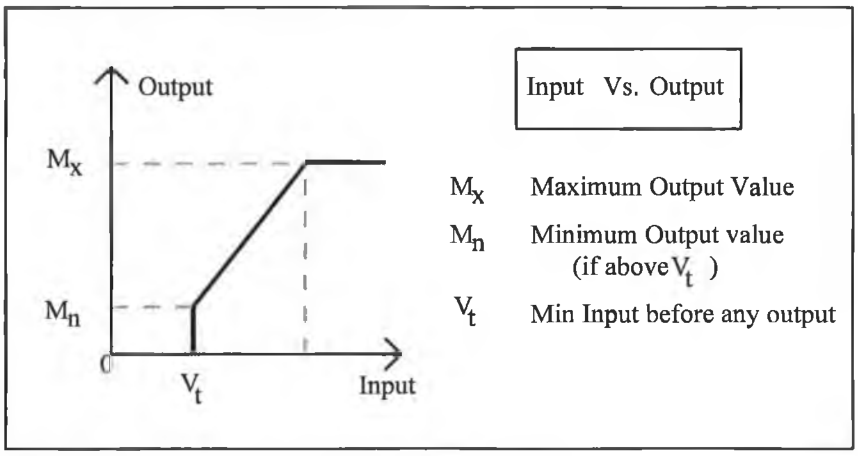


Figure 4.12 Non-Linear Nodal Input/Output Gain Characteristic

One of the most influential factors on the operation and simplification of the neural node's parameterisation and structure was not due to the real time aspect of robotic control at all. At the early stages of the project the neural node model was very similar to the nodal model used by Beer[2]. The network information, including the internal nodal parameters, was stored as an array of C structures made up of 38 variables (some combined within arrays). This high level of parameter variability offered a large degree of freedom to the network designer and hence made the design of the controlling neural networks by hand easier than it might otherwise have been.

Unfortunately, this high level of variability dictates a genetic search space whose size becomes unmanageable within the confines of the simple genetic algorithm operation. The population size, as already mentioned, was restricted to only twenty individuals. This was because the behaviour of each network was evaluated by hand. To stand any chance, within a reasonable time frame for the project, of allowing the SGA to converge on an optimum it was deemed vital to drastically reduce the size of this space. However, it was equally important not to over-minimise this variability. An over-enthusiastic reduction of variability could have actually impeded the operation of the SGA. Over simplification would trivialise the search. It could also have prevented the genetic operators from ever finding any kind of operational, much less optimum, neural network due to the smoothness of the genetic space being disrupted. Remember that a smooth search space is recommended as being essential to good SGA operation. A decision was taken to set the parameters of each type of similar node to the same value. This affected for example the threshold parameter, the minimum activation level and the maximum output levels to name a few. This decision was a difficult one as it altered the balance of power in the networks. It shifted it from dependency on the individual nodal characteristics for controlling the output, to dependency on the network morphology itself. In defense of this decision, it was hoped that the reduction in variability was not so severe as to prevent the SGA from still finding an optimum. It was hoped that it would overcome this reduction by compensating for the loss in parameterisation by experimenting more with the interconnections between nodes to achieve the same results. As it turned out, this decision seems not to have been badly guided, as the final results indicate.

Shown in figure 4.16 is a diagram of the C structure used to represent the neural node model prior to the simplification. It also shows the C structure representing the neural node model finally implemented in both the simulator and simulation software.

```
struct neuron
      {int buffer[5];
       int input[5];
       int outQtime;
       int inQtime;
       int minactlev;
       int last;
       int threshold;
       int gain;
       int NUM_INPUTS;
       int NUM_OUTPUTS;
       int weight[5];
       int connection[5];
       int nextnode[5];
       int nextspace[5];
      }     node[MAX_NODES];
```

a) Old node structure

```
struct neuron
      {int buffer;
       int input;
       int last;
       int weight[MAX_NO_OP];
       int nextnode[MAX_NO_OP];
       int output;
      }     node[MAX_NODES] = {0};
```

b) new node structure

Figure 4.13 Old and New Structures Comparison

4.5.4.4 Simplification 4 (Neural Node Update Routine).

The third simplification to the simulator and simulation software was caused by difficulties encountered with the input figure update routine. At the end of the first stage of the simulator software development, the software was operational. However, the code used to update the input figure was very contrived and very difficult to follow. After the two revisions already mentioned, this situation had become even more aggravated. It was decided that it would be better for if this very important routine were rewritten more clearly. It was decided to return to basics and use a difference equation approximation of the RC circuit of the new neural node displayed in figure 4.17.

The derivation of the input figure update difference equation is shown in appendix B. Beers differential equation is also shown in appendix B for comparison. The resultant equation used in the final simulator and simulation programs is:

$$V_c(t) = \frac{[C \times V_c(t-1)] + [\Delta t \times I_i(t)]}{[C + \frac{\Delta t}{R}]}$$

C: Models the capacitance properties of a real neurons cell membrane.
R: Models the conductive properties of a real neuron' s cell membrane
$V_c(t)$: Models the charge stired by a neuron at time *t*
$I_i(t)$: Models the input to a neuron at time *t*

Figure 4.14 Equation used to update node input value. (Vc in figure 4.19)

This change of routine simplified the program a great deal and made it a lot easier to understand. However it became much more difficult to design a network by hand due to the decrease in the number of nodal parameters that could be 'tuned' to achieve the desired network operation.

A further change made to the update routine was the incorporation of a buffer into the network node structure. The buffer was introduced to ensure that, during the update of the network's nodes, the input value used by the node being updated at time (t) was the output produced by operating on the input connections from connecting nodes at time (t-1). This buffer created a more structured and reliable neural network update routine.

4.5.4.5 Final Revision (Timing).

The final revision made was perhaps the simplest and least complicated. It was decided that it would be beneficial if the update time for any network designed was the same for all networks. The new specification was that it was to require the same amount of time to update any network regardless of the network's size or complexity. This was to make the simulator's operation more tractable, and make it easier to evaluate networks generated in the latter part of the project by the genetic algorithm. Also, and more importantly, this decision was based on the real time requirement of the robot's operation. Beer's model, as it was only used to control a simulated insect, could afford to allow the update time of the network to vary. Any variance would not influence the perceived behaviour of the simulated insect as the behaviour was automatically synchronised with the environment in which it operated. However, in dealing with a real robot controller, it would be unacceptable to allow the robot to 'pause' while its controlling artificial neural network was updated. It would not be beneficial to have the robot behaviour varying as a function of the number of nodes, the processor type or clock speed. The easiest way to achieve this is to force the artificial neural network to update at a fixed, real time rate, independent of the mentioned parameters.

This timing specification was realised and tested with the use of the MC68230 PI/T chip on board the MC68000 SBC. It required the programming[30,33] of a set of clocked registers on the PI/T to an initial value (derived from the clock speed of the SBC and the network update time specification). Then all that was involved was the polling of a single bit on the PI/T's status register, which changes from 0 to 1 when the PI/T's clocked registers hit zero. The clocked registers then cycled back to their initially programmed value and the process was re-initiated.

Within each cycle the simulator updates the network once and then simply polls the PI/T bit position mentioned above until it changes. Then the cycle begins again.

4.5.5 Node Types

So what was the resulting node structure like after all these changes? In this section the different node types used will be detailed. There were three types eventually used: Normal nodes, Generator nodes and Output nodes. The dynamics and structure of each of these are very similar but the differences affect the operation of the network considerably.

4.5.5.1 Neural Node Description.

The final state of the neural node as implemented in the latter stages of the project remained quite consistent with the node model used by Beer. The internal dynamics remained the same in essence. The output gain profile (see figure 4.15) remained the same and the capacitance modeling was also retained, albeit a difference rather than a differential equation implementation. The internal conductance parameters as per Beer's model (see figure 4.18) were not included. For comparison the two nodes are shown side by side in figures 4.18 and 4.19.

The neural node can receive any number of inputs. These inputs take the form of being weighted between the values of 0 and 255. This sum is then delayed by the neural network update time (t). This sum is then used within the difference equation representing the RC circuit. The result of this (V_c in figure 4.18) is used by the output gain section to calculate the output as per the gain characteristics illustrated in figure 4.15. This results in an integer figure in the range 0 to 255. This figure is then used by the output section. The node can have up to a maximum of ten outputs. Based on the genetic parameters for inhibition, the number is made negative for those connections which should be inhibitory. Finally, the weighting of the value is based on the distance[13] to the next node.

[13] The distance referred to is the distance on the growth grid established when the networks were grown.

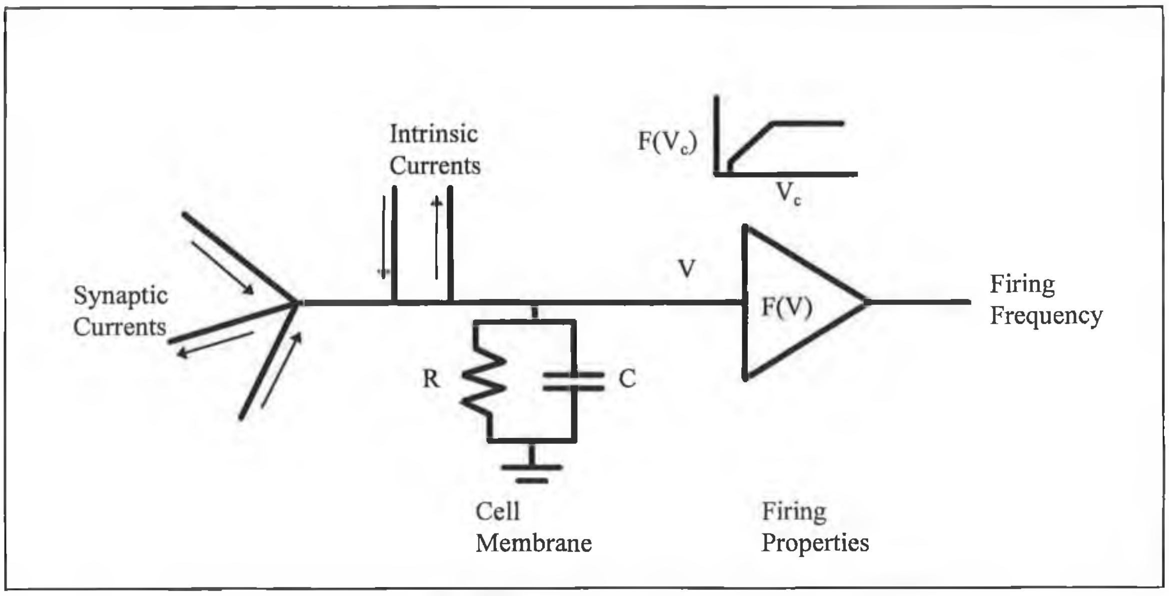


Figure 4.15 Neural Node structure implemented by Beer

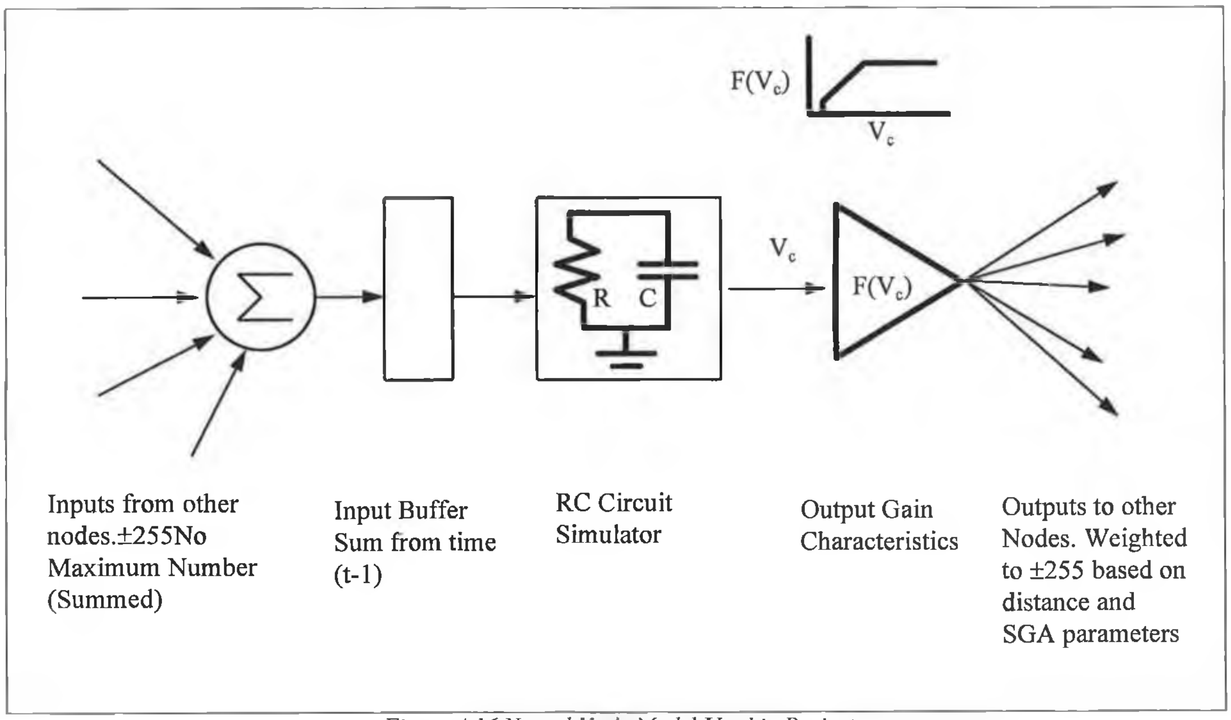

F(V_c)
V_c
V_c
R C
F(V_c)
Inputs from other nodes.±255No Maximum Number (Summed)
Input Buffer Sum from time (t-1)
RC Circuit Simulator
Output Gain Characteristics
Outputs to other Nodes. Weighted to ±255 based on distance and SGA parameters

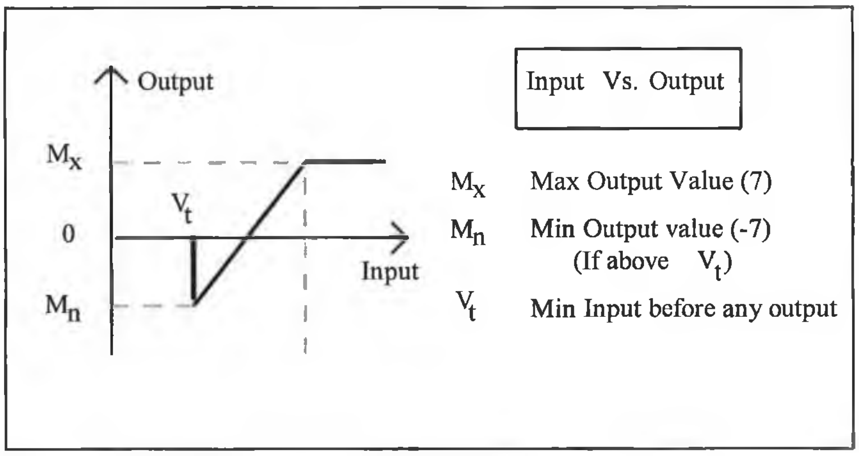
Figure 4.16 Neural Node Model Used in Project

4.5.5.2 Generator Nodes

Generator nodes were introduced to compensate for the lack of spontaneous activity which could be displayed by the network. This was due to the omission of the internal currents implemented in Beer's neural node model for the reasons described in section 4.5.4.1. Essentially, the generator nodes continually produce an output of 255 regardless of input.

4.5.5.3 Output Nodes

Output nodes behave exactly like normal nodes except for two crucial differences. Obviously they have only one output. More importantly however, their gain characteristic is shifted by 50% so that they can produce a bipolar output. Also the output figure is normalised to within ±7 to work correctly with the SBC to robot interface. The gain characteristic is shown in figure 4.20.



Output
Input Vs. Output
M_x
V_t
0
M_n
Input
M_x Max Output Value (7)
M_n Min Output value (-7) (If above V_t)
V_t Min Input before any output

Figure 4.17 Gain characteristic for Output Nodes

4.5.6 Summary

In this section the neural networks and the manner in which they were used to implement robotic control were described. The individual components of the neural networks were detailed. This included the individual node types implemented as well as the connection mechanism used between them. The manner in which the networks interfaced to the real world was also described. So what are the final conclusions?

The networks were, in the end, quite dissimilar to the type of networks originally envisaged. Due to implementation difficulties raised by both the real time considerations for robotic control and the time frame available for the traversal of genetic spaces by the SGA, the components of the network were significantly reduced in complexity. The individual nodes were the single area most influenced by these considerations. The internal dynamics of the neural nodes were altered in many ways. The internal current modeling was removed. Also, the manner in which the capacitance properties of real neurons operate, over the scope of the full network, was simplified. This was achieved by the removing some of the individual characteristics of each node in the network. Instead each node adopted the exact same capacitance properties by setting the internal variables equal to some network spanning constant.

However, the neural networks, as will be revealed in chapter 6 did succeed in performing. They satisfied the criteria imposed on them for real time robotic control despite the simplifications made. For that reason, and that reason alone, I feel that the decisions made over the course of the project were fully justified. As well as the operational requirements imposed on the networks being satisfied, the considerations for operation under Brooks' subsumption architecture model were also satisfied.

Although this remains untested, the manner in which a superior behavioural network could subsume control is straightforward enough to deserve a textual description rather than necessitating a test run. A superior behavioural network could subsume control by forming connections with, and only with, the input and output nodes of the underlying network. In this manner a network such as that which, for example, supported a simple visual processing operation could use its own input to stimulate, and inhibit, the underlying network's input and output nodes. This could cause the underlying network to behave as the upper network desired. This would still allow the underlying network to exhibit the same characteristics as before if the upper visual network to be damaged or the robot blinded for example. The robot would thus adapt to its new situation by using what it already 'knows'.

There has been much mention in this section and others of the robot's successes in the course of the project. However, how were these successes quantified and qualified? After all we are dealing with behavioural issues. What constitutes good behaviour and how can it be measured? This will be discussed in the next chapter.

5. Behavioural Evaluation

5.1 Introduction

As has already been mentioned, one of the cornerstones of the project documented in this thesis is the subsumption architecture model for the development of machine intelligence. Subsumption architecture relies on a behaviourally decomposed design structure. Each level of the architecture uses the previous levels' behavioural patterns in conjunction with its own behaviour in order to increase the behavioural complexity of the overall robot. Therefore, the evaluation of good behaviour at any stage is vital to the later development and enhancement of the robot. In this section the manner in which the behaviour of the robot was evaluated will be explained. The rules governing the evaluation will be detailed. There will also be a discussion of what was perceived as 'good' behaviour at the start of the project.

In the context of using simple genetic algorithms and neural networks the evaluation of good behaviour becomes even more critical. In a functionally decomposed robot, any anomalous behavioural patterns can be easily identified. Their solution can be the replacement or refinement of a single, and more importantly easily definable, circuit or software section. Neural networks, as applied to this project, are grown as a complete entity and the many 'sections' of each network are intertwined. Therefore it would involve a more complex and time consuming process to identify and tweak by hand the network's internal parameters. Also, using a simple genetic algorithm to derive a controlling network further exacerbates the situation. As applied to this project, the

SGA implementation is devoid of any internally coded objective function (see section 2.4). The objective function used to guide the SGAs traversal of the genetic search space is the behaviour exhibited by the robot. It is essential that any particular behavioural pattern exhibited by the robot under the influence of the neural network is always awarded the same score. In the next section the types of behaviour rewarded will be described and justified in the context of the early behavioural goals of the project.

5.2 What is good behaviour?

The introduction to this section emphasised the importance of a reliable mechanism for behavioural evaluation for this project. The behavioural goal of the project was to implement a robot that could wander around its environment and not become stuck. The robot was to achieve this using six touch sensors and two bi-directional motors to drive its wheels. But, what defines good behaviour? A number of assumptions were made about what the implications of good behaviour would be, in the context of this project. It was desired that the robot would move spontaneously when its evaluation began. Any cyclical behaviour[14] would be discounted. The operation of each of the sensors would be evaluated independently. Also, any movement on the part of the robot that could be considered exploratory would be rewarded. The environment used to evaluate the robot is described in section 3.6 and was designed to test the robot's ability to remain unstuck in a number of specific ways.

Area A (see figure 3.6) was a large area used to ensure that the robot did not simply move backwards and forwards in a straight line, or go around and around in a tight circle. It was constructed to ensure that the robot did not demonstrate some form of cyclical behaviour that would prevent it ever reaching out to the borders of its environment. Area B was used to see if the robot could maneuver its way out of a tight cul-de-sac without being able to turn around. Area C was used to see if the robot could maneuver its way out of a confined space by finding, and using, the narrow entrance

[14] Moving in a tight and unbroken circular pattern for example.

through which it entered the area. Area D was used to see if the robot could combine the behaviours highlighted by areas B and C. It was hoped that the robot could find the exit while reversing out of the area. Area E was used to evaluate the robot's ability to handle an acute angle within an area.

Overriding these simple goals, in accordance with the first stages of Brooks' subsumption architecture, it was hoped that the robot would move all around the environment and seek out these different areas as well as conquering them. In particular perhaps, exhibiting some form of edge following behaviour.

These perceived goals are simple ones that could be achieved (rather simply one imagines) with the use of a standard functionally decomposed design framework. It was felt however, that these goals were sufficient to test the overall design framework and the validity of the thesis. If these goals were achieved, then the potential would exist to continue to work in this application of machine intelligence. In the next section, the specific rules used to calculate the robot's behavioural 'score' for the SGA will be described.

5.3 The Rules

It was essential during the evaluation of the robot's behaviour that the assignation of scores to particular behavioural patterns remained constant throughout the evaluation period. Any deviation could have had disturbing effects on the operation of the SGA. Therefore, a number of maintainable rules were devised in order to evaluate the particular behavioural score of each robot.

A program was written to ensure that each area of operation was evaluated correctly. This program is called GENETIC.C and is contained on the disk accompanying this thesis. The original intentions for evaluation are listed below for clarity.

1. The robot's behaviour shall be evaluated over a four minute period.
2. The robots initial position and direction will be in the centre of the environment and the initial orientation shall be held constant (as much as possible) for all the robots. (Exact positioning is not necessary, or indeed possible.)
3. The first behaviour that shall be looked for is characterized by two parts. Firstly, the robot moving around the environment while not getting stuck and secondly, the robot will not overly twist its 'umbilical cord'. (i.e. equal distribution of left and right directional movement shall be rewarded.)
4. Each robot is evaluated on a 100 point scale.
5. Subtractions are made from the 100 points as follows to calculate the robots behavioural score.

Failure to move initially.	- 10
Failure to change movement when stimulated. Seven marks each for operation and direction.	- 14 per sensor
Becoming trapped in the environment.	- 10 per occasion
Twisting the wire thus preventing the robot turning.	- 10 per occasion

Figure 5.1 Basic Behavioural Scoring

6. All individuals shall have a minimum score of 10.
7. The sensors are depressed in order of precedence. It is perceived as more important that the front sensors operate rather than the rear.

A) Front sensors together.	(7 marks)
B) Left front sensor/Right front sensor.	(7 marks each)
C) Rear sensors together.	(7 marks)
D) Left rear sensor/Right front sensor.	(7 marks each)
E) Left side sensor/Right side sensor.	(7 marks each)

Figure 5.2 Sensor Activation Scoring

8. Should the robot be stuck at any time, it shall be replaced in the centre of the environment facing 45° clockwise of its initial position. From that position its evaluation shall continue.

These rules were generated over the course of the development and debugging of the overall environment. A number of runs were made at the early stages of the project to validate the behavioural rewarding scheme used. Eventually, the situation detailed above seemed to cover the majority of behavioural situations that it was desired to reward and penalise.

5.4 Summary

In this section the rules used to evaluate the behaviour of the robot during the operation of the SGA were detailed. The rules chosen were simple and left little room for 'personal' preference or error on the part of the robot's behavioural evaluator. It was explained that the rules were designed in such a way as to attempt to implement the first stages of Brooks' subsumption architecture design methodology. The rules were used as they are documented above in two independent 'runs' of the SGA. Considering the results achieved and the best behaving network discovered, the choice of rules was vindicated to a large degree. In the next chapter of this thesis the results obtained will be detailed.

6. Results

6.1 Introduction

There has been continual reference in this thesis to the results obtained. The results have been used a justification for many of the decisions taken at every stage of the framework development. The rationale behind some of the decisions could be considered as tending towards arbitrary. Where applicable however, all decisions were guided by the notion of trying to replicate accepted natural system processes. The justifications for these decisions was based on the desire to follow an ideal, and by making as few compromises as necessary to allow a real time and maintainable implementation. The decision to 'grow' the neural networks rather than use a network 'blueprint', for example, was not used because it was believed that it necessarily offered a greater chance of success. A growth algorithm was used because it identifies more closely with the processes evident in real life.

The 'life' approach was used because it was interesting, and also because it seemed to offer an intuitively satisfactory approach to the development of a low level machine intelligence. It was desired that Brooks' behavioural approach (the original inspiration for the project) be brought closer in application to the real life processes evident all around us in the real world. Brooks' implementations involved programmed solutions. Real life however is not programmed. Indeed, it could be argued that by using a programmed solution, Brooks' solution falls victim to some of his own arguments. One of the tenets of his argument [4] is the rejection of representational approaches to the development of artificial intelligence. The potential intelligence of a final programmed or representational solution is limited by the intelligence of its creator. It would be much more satisfactory, from a theoretical point of view, to develop a framework capable of producing an intelligence that would be, if not greater than our own, at least different. This different intelligence could offer a perspective on our own form of intelligence which could make it easier to understand ourselves.

In hindsight, the attempt to implement and combine the different natural systems used in this project seems rife with potential disasters and laced with unforeseen problems. However, at the end of it all, results were obtained - results which appear to justify and vindicate the risks taken in using this approach to the development of machine intelligence. In this section, these results and the manner in which they were obtained will be documented. Their importance, in the context of justifying the decisions made in this project, cannot be over emphasised. However, the results do not vindicate every decision made in this project. Indeed, they throw open some quite serious implications for the future of this development framework.

6.2 *Two Runs*

The testing and tuning of the design framework mechanism took place over a period of approximately four months. The first months of this process were used to test the integration of the framework as a whole design process. Certain issues became apparent over the period of this testing involving the integration of some of the changes already documented in this thesis. The changes made to compensate for the errors detected in the desired operation needed to be tested in order to validate the framework at each stage. Eventually, a point was reached where the framework was considered satisfactorily stable to allow a proper evaluation run to take place. Two runs of this 'final' configuration were made over a period of about 6 weeks. The results of both these runs will now be detailed.

6.2.1 Run 1

A graph of the results obtained in the first run are shown in figure 6.1. This graph represents the convergence demonstrated by the SGA's operation over a period of forty generations involving a population of 20 genotypes. As can be seen the SGA does appear to converge to an optimum in the search space. However, it is also apparent that the best behaviour found for a single robot occurred in generation 25. The SGA continued to converge however to a different point in its search space. The highest point of convergence occurred with a correlation factor of 84% between the genotypes. At this point the maximum scored by any of the robots contained within that particular generation was only 63. It appears that the SGA did indeed find a point at which the robot 'behaved well' (see chapter 5), scoring a value of 78 points on the evaluation scale. However it also appears that the increase in convergence levels fluctuated and the point was lost. Perhaps if the run had continued for a greater period of time the optimum would have been relocated. So what could have caused this detection and loss? A number of factors may have contributed to this effect. The first, and most obvious one is that the level of mutation employed by the mutation function may have been excessively high. An excessively high mutation level can significantly disrupt the operation of the SGA. Alternatively, the mutation function could have simply hit the genotype in a 'soft spot' causing a level of disruption that was detrimental to the progression of the SGA run. This could indicate that the schema populating the SGA's genotype may be excessively long. A short schema is recommended by Goldberg in his discussion of the SGA [14].

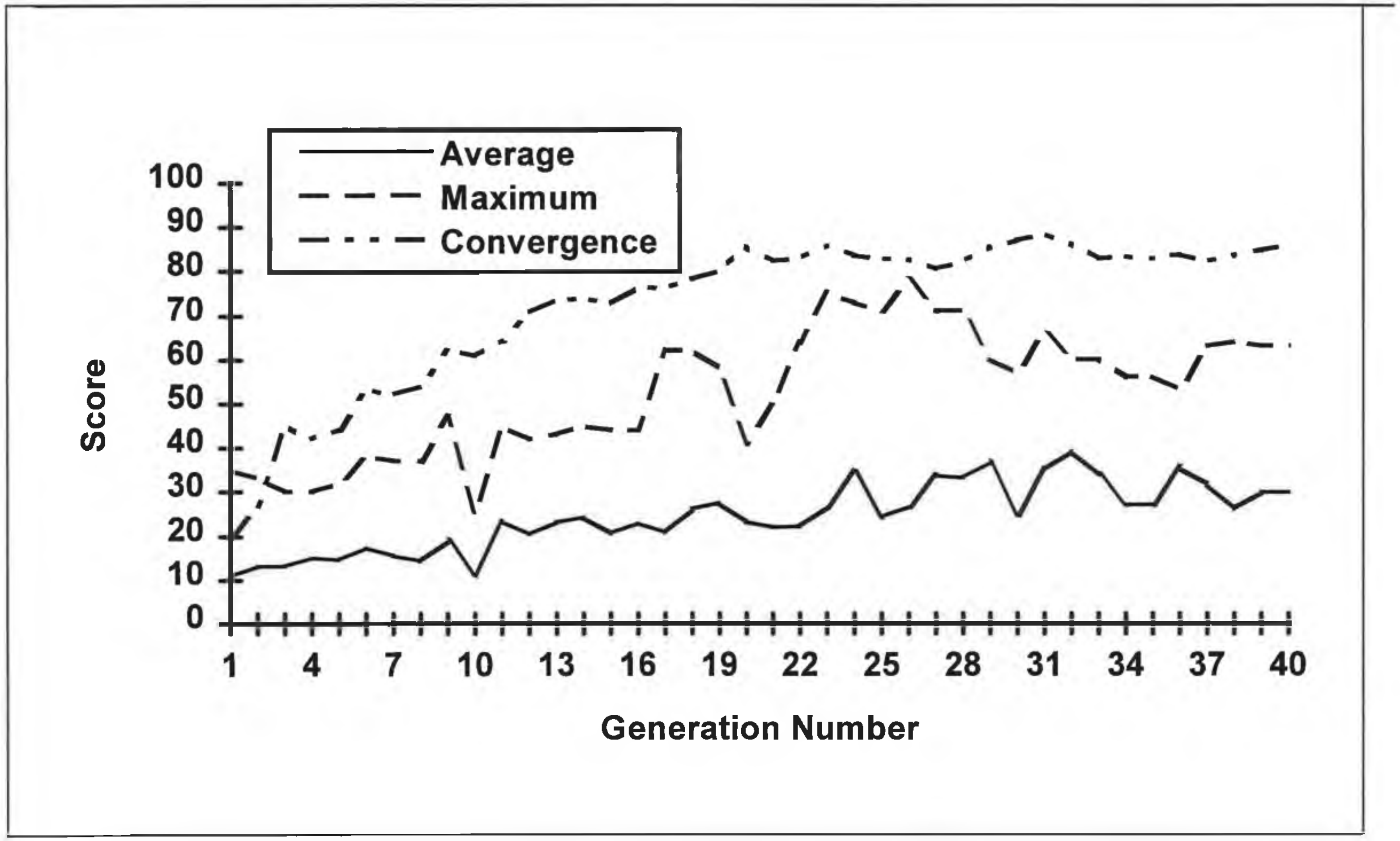


Figure 6.1 Operation of SGA for Run 1

However, despite the loss, I feel that it more worthwhile to examine what was achieved by the framework rather than emphasising its difficulties. At generation 25, a robot controller was produced which scored a value of 78 on the evaluation scoring mechanism. The behaviour of the robot was as follows. The robot being controlled by the grown neural network moved spontaneously in the environment. It traveled in a straight line until it reached the boundary wall at position Z in the environment (see figure 3.6). The robot then proceeded to 'explore' the environment and found its way into and out of almost all the areas in the network. The robot, while exploring, exhibited a form of edge-following behaviour moving in an anti-clockwise direction around the environment.

The robot demonstrated some quite interesting behaviour when it appeared to be trapped. When the robot was placed in the area designated by the letter 'B' in figure 3.6, facing the end wall, the robot moved towards the end wall, using the right hand edge of the cul-de-sac as an indicator. When it got to the end, both front sensors would be activated and the robot would attempt to back away from the wall moving in an clockwise arc. See figure 6.2.

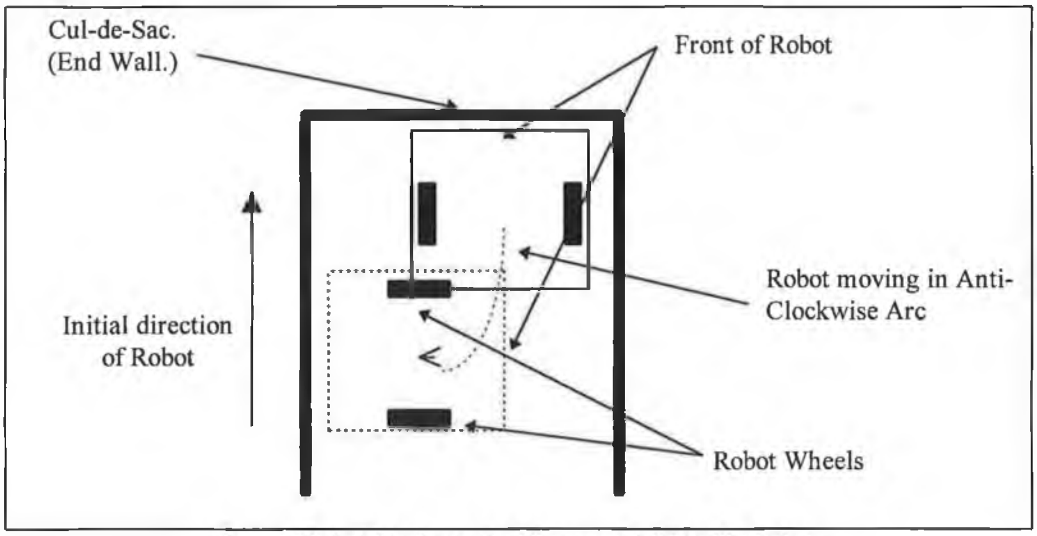

Cul-de-Sac.
(End Wall.)
Front of Robot
Initial direction of Robot
Robot moving in Anti-Clockwise Arc
Robot Wheels

Figure 6.2 Anti-Clockwise Arc Movement Exhibited by Robot.

The robot would detect its back left sensor being activated and would move forward again. The front left sensor would come into contact with the wall and the robot would circle around until both front sensors were activated once more. Again, the robot would move backwards in a clockwise arc until its back left sensor came into contact with the wall once more. This time however, the sensor activation would not cause the robot to move forward again. The robot's behaviour suggested that it "remembered" that it had recently had hit the wall. The robot would continue to press against the wall and the motors would continue to drive the robot back. The robot would thus swing around until both rear sensors were activated. The robot would then move forward again until both front sensors touched the side wall. The cycle would begin again and the robot would maneuver itself right around until it was facing 180° from its initial direction. The robot then exited the cul-de-sac.

6.2.2 Run 2

The second run of the SGA did not produce results that were quite as good. The graph of the genetic scores is shown in figure 6.3. The SGA converged on some optimum but unfortunately the behavioural scores did not reflect the same promise exhibited by the first run. The maximum score of any robot was just 26 points and the average never exceeded a value of 13 points. The maximum level of convergence was at the point where testing discontinued with this run. It assumed a value of 86% correlation between genotypes. Perhaps if the run were continued the SGA would have removed itself from the local optimum it had found. The effect of mutation an reproduction can be seen quite visibly on the graph at generation 20 where the convergence figure fell by 10% over a single generation. However, this change in average genotype characteristics was not sufficient to drag the SGA away from the genetic hill that it was climbing.

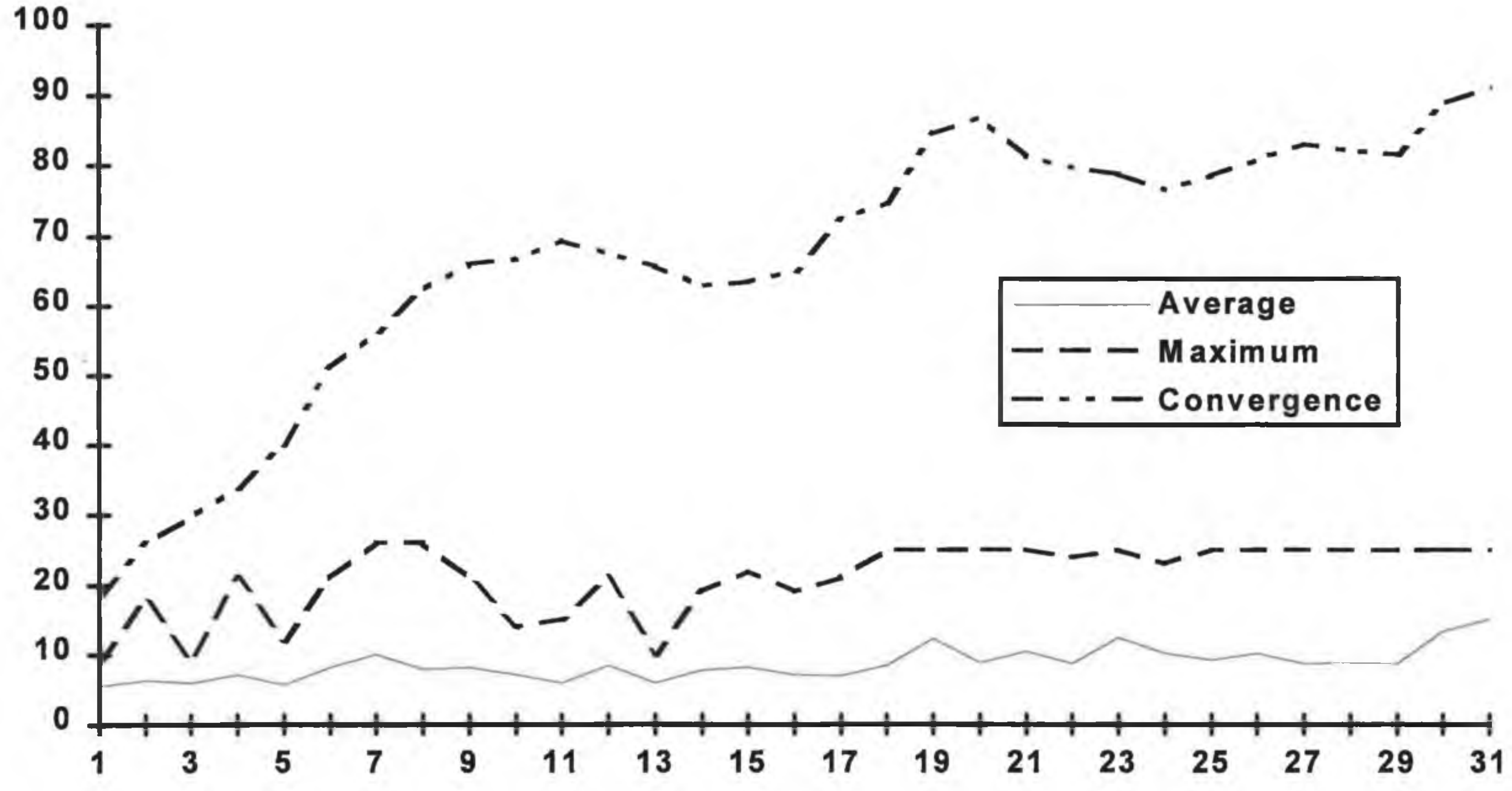


Figure 6.3 Operation of SGA for Run 2

6.3 Results Conclusions

So what conclusions can be drawn from the two runs of the SGA. I think that it is necessary to consider the results in two separate categories. The first of these categories is the operation of the SGA and the second is the operation of the overall network.

6.3.1 Simple Genetic Algorithm

With regard to the SGA, it can be seen from the first and second run that the algorithm does converge on an optimum in the genetic space. However, it is also obvious from the results of run 2 that the parameters of the SGA need to be re-analysed. It may be even beneficial to analyse the use of a simple genetic algorithm at all. It may be entirely possible that this type of genetic algorithm implementation is flawed in application to neural network control. It works certainly for simple and well defined problems but perhaps the complexity involved in neural network design is beyond the scope of this simple algorithm. The scope of genetic algorithms and their applications extends far beyond the SGA implementation. This is portrayed in Goldberg[14] where many different GA implementations are mentioned and described. It is therefore conceivable that a different implementation of the GA may have been more suited to the task in hand although this was never investigated.

6.3.2 Overall Framework

On the positive side however, a neural network was found in the genetic space that did satisfy the requirements of the project. The robot using the network described in section

6.2.1 successfully traversed its environment in a manner approaching the optimum desired manner.

7. Conclusion

This thesis has charted the design of an evolutionary framework for the development of low level machine intelligence. The framework was successful in many respects. Principally in that it was almost fully successful in achieving its primary functional objective. This objective was the development of a neural network that allows a real robot to move around in a real environment. The goal of the robot/network interaction was to prevent the robot from becoming stuck. By doing so, I feel justified in saying that the use of a natural systems framework is a viable method for the achievement of low-level machine intelligence. However, a number of issues still remain unresolved. In this conclusion I will go through each of the main areas in turn, both theoretical and functional, and consider the problems encountered with each while attempting to provide possible solutions.

The reader should be aware that this project has only taken a small, but important first step in the development of machine intelligence using natural systems models. It would be fair to say that the final state of the project has generated as many possibilities and questions as it first set out to answer. More questions however, is a much more satisfactory end point than a dead end.

7.1 Subsumption Architecture

The concept of a behaviourally based control decomposition for the eventual development of a high level machine intelligence is at the heart of this thesis. However, it was never within the scope of this project to verify the approach itself. The only true validation of the subsumption architecture approach would be the creation of a high level machine intelligence. By this I mean the creation of an artificial mind that can learn, adapt and reason by utilising fully and adaptively a pre-existent physical framework, and that behaves autonomously and successfully within a hostile, dynamic environment. Attempting to achieve a lower level of behaviour, such as the type within this project, demands a level of faith that the higher level goal is achievable. Therefore, in all further discussions it is assumed that the incrementally designed behavioural decomposition structure is a viable method for attaining the long term goal of machine intelligence.

The essence of a subsumptive architecture, within the context of the achievement of a learning, adaptive machine intelligence, is such that in order to support such an adaptive intelligence on a physical structure then the lower level controllers must be fully functional and secondly, must be of such a type that they facilitate the subsumption of their lower level network capabilities by higher level networks.

Certainly, the lower level networks formed to date do not learn or adapt in the sense of displaying on-line structural or parameter variance within the controlling neural network. However, the networks produced for the goal of enabling a real robot to wander around a real environment have demonstrated themselves as being able to deal successfully and adaptively with that environment. Also, the physical structure of the networks allows the idea of behavioural capability absorption to be applied easily.

To illustrate this idea, consider attempting to design a visual processing control network, allowing long distance visual goals such as finding the corners of rooms. The visual processing network would sit on top of an existent tactile stimulation reaction network such as the small neural networks designed and demonstrated in this thesis. The structure within which the tactile networks are grown is such that it is very achievable to 'sit' another network on top of it. This can achieved by allowing, for instance, the higher level visual network to connect only to the input sensor nodes and output motor nodes of the lower level network. The visual network can then subsume control by having eight outputs connecting to the six input nodes and the two output nodes of the tactile network and it could generate behaviour in the robot by simulating the input patterns to the tactile network that result in the robot responding in the manner desired by the visual network.

Consider, for example, a situation where the robot sees a corner of a room and wants to get there. Using the subsumption approach the visual stimulation could result in the rear input sensor nodes of the tactile network being stimulated, not by physical contact with a physical obstacle, but by the higher level network, and moving forward toward the goal. In a sense, the higher network is ‘fooling’ the lower network into believing that its rear sensors were being activated and thus using the established reaction of the tactile network to move forward.

7.2 Computational Neuroethology

The concept of modeling the neural structures of existing animals such as insects is, intuitively, a very sensible one when tied in with the idea of a subsumption architecture. However there are drawbacks. The main drawback is that even for very simple insects the structure of a single neuron is quite complex and a large network of accurate neural models would soon be beyond the simulational capabilities (in real time) of many of today's largest and most powerful computers. This is of course due to the essentially serial nature of modern computers. The ideal for simulation of a neural network would be a massively parallel computer with an individual processor for each neuron. However this is not viable either technologically or financially within the scope of almost every research project. This means that serial computers have to be used.

As a result of the imposed limitations, the neural models lose the finer points[15] of their operation which of course affects the neural operation. The models used during this research were even simpler models of a simplification used by Beer of a real neural node. This simplification allowed real-time operation of the robot. Also, the decision to omit random noise from the neural calculations makes the neural network operation deterministic. This enables all results to be repeatable, a facet of design which is essential at any stage of a project but I feel particularly so at the early stages of a large project such as the development of machine intelligence.

[15] The intrinsic currents implemented by Beer [2] for example.

Any simplification brings into question, of course, the validity of the claim to be using a biologically plausible neural model. We do not truly understand the complexities involved with operation of simple biological neural structures, and much less so the neural structure of the human brain, whose sentient characteristics we are attempting to model, so how can we simplify something we do not truly understand? The complexities and internal noise present in biological neural networks are an inescapable fact so perhaps we are discarding on a neural node level the most essential characteristics for the achievement of machine intelligence.

Unfortunately, due to the limitations imposed by the available technology for this research, there is no option but to simplify. However I believe, that within the scope of this project, this has been verified to be an acceptable path to take for the design of networks exhibiting lower level behaviours. It has been implemented and demonstrated to produce an operational robot controller that allows the subsumption architecture to be applied. Despite the relative success however I feel that more could, and will, be achieved by using a more biologically plausible neural model operating on the best that technology can offer.

7.3 Computational Embryology & Genetic Algorithms

In their application within the context of this project the concepts of computational embryology[9,10,11] and genetic algorithms[14,19,34] are entwined within each other to such an extent that a discussion of one without reference to the other is not possible. What is being modeled in order to grow the neural network is not just a series of natural systems models but an entire process of development, growth, and long term genetic adaptation of a neural network to its environment. Therefore they shall be discussed together.

The computational embryology and genetic coding approach again seems to be, intuitively, a correct choice. I justify this decision by drawing on the essence of the project which is to apply, as much as possible, natural models to develop the controlling neural networks. Nature has already shown us that the use and variance of DNA coding, which it uses to allow reproduction and species adaptation, works.

Computational embryology is a simplified and digitised equivalent of the DNA decoding which nature uses. However, the DNA decoding process in nature is even more complex and mysterious than the networking structure and inter-operation of individual neurons in the biological brain and nervous system. Therefore, in exactly the same manner as the use of simplified neuronal models and network structure implies, the validity of using any simplification is called into question. But again, due to technological and knowledge limitations, simplification based on existing knowledge and experience is the only way forward.

It was a conscious decision to use a development and evaluation routine and it has demonstrated some success in application. Also, by using a parameterised growth process the networks produced may do something not previously considered, but useful when discovered, while still operating within the framework of an accepted goal. It also allows the smaller schemata of growth parameters to be used rather than large schema which would be involved in the use of a blueprint network coding. There are problems of course in the use of a development routine rather than the use of a network blueprint.

Firstly, analysis is more difficult to do than it would be using a blueprint coding, but at a lower behavioural level this is not a large problem because the networks produced are quite small (max. 50 nodes in this application), allowing a visual analysis of the network structure. As network complexity increases however, with the development of higher level behaviours, visual analysis will become impossible to do. This inability to analyse leads to the deeper question of whether or not it makes sense to design something which we cannot easily understand. Are we attempting to implement or to understand intelligence? For this project, implementation was the objective.

Secondly, as a result of the coding, the interaction of individual schema within the genetic coding becomes less defined, thereby increasing the complexity and non-uniformity of the genetic search space. This causes the use of a genetic algorithm as the chosen optimisation routine to be called into question as genetic algorithms require a smooth search space to enable them to behave in an optimal fashion. Indeed this is demonstrated by comparison of the two runs of network production listed in the results section. The first run produced the best behaviour in a single individual but the next did not produce any individuals (within the scope of the number of generations iterated) that produced a comparably successful individual. Within the operation of the SGA the genetic algorithm should converge to the same genotype. This may be due to a premature termination of the generation run but may be more symptomatic of a fault with the chosen coding and development routines.

I would be inclined to choose the latter explanation and for that reason would recommend a number of changes to be made to the overall design framework should work be continued based on the existing results. Firstly, a change from purely binary coding to a gray-scale coding or some other coding which results in equivalent numeric weight being attached to each binary position in the genetic coding thus preventing mutation of a single bit causing as much genetic diversion as is possible with standard binary weighting. Secondly, the use of scaling factors[14] and multiplication factors[14] to allow greater control of the genetic algorithm operation preventing as much as possible premature convergence of the genetic algorithm. Thirdly, an increase in the precision of the genetically coded parameters to further smooth out the genetic search space allowing the genetic algorithm to behave more optimally.

7.4 Other Issues

The most contentious issue encountered over the period of the research is involved with the question of: how exactly does one quantify good or bad behaviour? This is a critical issue because if good behaviour cannot be quantified then automation of the entire behavioural evaluation, genetic algorithm operation and further network growth becomes impossible. As the research outlined in this thesis progressed, it got to the stage where a decision had to be made regarding whether or not to automate. I believe that it is possible to quantify behaviour to some degree and that automation is possible at both higher and lower operational levels. This belief is prompted by the fact that even with manual evaluation it is essential to create some form of rule base by which to evaluate so that the genetic scoring is consistent with robotic behaviour rather than testers' moods. However, automating the evaluation procedure would require the development of simulation software as well as adjustments to the hardware and interfaces set-ups and constructions. This was impossible within the available time scale of a single masters project so it was decided to forge on with manual evaluation and attempt to gain some results which could justify the effort involved in automating the evaluation procedure should the project be continued at some stage. This evaluation was based on a visual interpretation of the robot's behaviour by a person, with the genetic scores award process being automated to a degree by some rather simple software but ultimately decided by the person testing the networks' operation.

This, as compared to what could be achieved with a good automated procedure is unsatisfactory in the long term. It is too time consuming and too unproductive from the human tester's viewpoint to be a viable practice.

Although I am certain that automation needs to be achieved to progress, automation is not all good. The chief problem with automation is that it generally implies use of simulation - simulated robots in simulated worlds. This, I am certain, is not the correct path for the achievement of machine intelligence either. A compromise between the two extremes may be the best way forward. Since the goal is to develop a robotic system that can survive in the real world, it is essential that the networks which appear to be working in the simulated world be tested frequently on a robot in the real world. This serves two main purposes. Firstly it verifies that the simulation software being used is operational and secondly it allows continual calibration of the simulation results with the real world results as well as fine-tuning of the evaluation procedure itself.

7.5 Final Conclusions

The project came close to its primary goal of developing a reactive tactile real-time neural network robot controller operating in the real world. Its major success lies however in its highlighting of the areas that need to be addressed and solved in order to make the overall system more productive and stable:

1. Further consideration be given to the network growth routine, preceded by a more detailed study of existing biological neural network growth characteristics.
2. A more in depth analysis and potentially the implementation of a different form of the Genetic Algorithm to improve its operation and smooth out the genetic landscape allowing it to operate more optimally.
3. Combination of the GA it with an *automated* evaluation routine allowing larger numbers of generations to be iterated with perhaps greater success in network design.
4. Review of the controlling hardware to increase the speed of operation allowing more complex artificial neural networks to operate.
5. Generation of an environment and robot interaction simulation. This would allow a more speedy evaluation of networks. These networks would necessarily be calibrated frequently against the real-world operation of the real robot.
6. Using the above changes (and assuming success in designing a fully operational reactive tactile network), attempt to design a visual perception neural network layer that will utilise the capabilities of the lower level tactile network.

In conclusion I feel that the application of natural systems models to the production of a viable design framework that generates behaviour controlling neural networks for real-time, real-world robotic control has been demonstrated. I also feel that the design framework has the potential, using the best of available technology and biological systems knowledge, to generate more biologically plausible neural networks that in application may eventually place the goal of machine intelligence in sight, however far away that holy grail of computer science truly is.

8. Bibliography

[1] Barkakati, N. "The Waite Group's Turbo C Bible", Waite Group Inc. Macmillan Computer Book Publishing Division. (1989).

[2] Beer, R. "Intelligence as Adaptive Behaviour: An Experiment in Computational Neuroethology", Academic Press. (1990)

[3] Brooks, R. "Achieving Artificial Intelligence Through Building Robots", AI Memo 899, MIT. (1986).

[4] Brooks, R. "Intelligence without Representation", Artificial Intelligence 47, Elsevier Science Publishers (1991) 139-159.

[5] Brooks, R. "Elephants Don't Play Chess", Robotics and Autonomous Sytems Systems 6, Elsevier Science Publishers B.V. (North Holland) 3-15 (1990).

[6] Cliff, D., Harvey, I., Husbands, P. "Incremental Evolution of Neural Network Architectures for Adaptive behaviour", Cognitive Science Research Reports CSRP 256, University of Sussex, School of Cognitive and Computing Sciences. (1992 - Jan)

[7] Cliff, D., Harvey, I., Husbands, P. "Explorations in Evolutionary Robotics", Adaptive Behaviour, **2**(1), 73-110. (1993).

[8] Cliff, Dave T. "Computational Neuroethology - A Provisional Manifesto", Cognitive Science Research Reports CSRP 162, University of Sussex, School of Cognitive and Computing Sciences. (1990 - May)

[9] Dawkins, R. "The Blind Watchmaker", Penguin Books. (1988).

[10] Dawkins, R. "The Evolution of Evolvability", Artificial Life, SFI Studies in the Sciences of Complexity, C. Langton Editor, Addison-Wesley. (1988).

[11] Dawkins, R., "Universal Darwinism", D.S. Bendalli (Ed), "Evolution from Molecules to Men", Cambridge University Press, (1983).

[12] De Jong, K. A., “An Analysis of the behaviour of a class of genetic adaptive systems”, Doctoral Dissertation, University of Michigan, Disertation Abstracts International, 36(10), 5140B. (University Microfilms No. 76-9381), (1975).

[13] Kenerney Anita, “An Autonomous Robot”, Undergraduate Thesis, School of Electronic Engineering, Dublin City University. (1988).

[14] Langton, Christopher G., “Artificial Life”. Artificial Life Volume VI: Proceedings of an interdisciplinary workshop on the synthesis and simulation of living systems held in September, 1987, in Los Alamos, New Mexico, Addison-Wesley Publishing Company, Inc., Redwood City, California, (1989).

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

[16] Harvey, I. "Species Adaptation Genetic Algorithms: A Basis for a Continuing SAGA", Cognitive Science Research Paper 221, University of Sussex. (1992).

[17] Harvey, Inman. "Evolutionary Robotics and SAGA: The case for hill crawling and Tournament Selection", Cognitive Science Research Paper, Serial No. CSRP 222, University of Sussex. (1992a).

[18] Harvey, I. "The SAGA cross: The mechanics for Recombination for Species with Variable Length Genotypes", Cognitive Science Research Paper, Serial No. CSRP 223, University of Sussex. (1992b).

[19] Harvey, I., Husbands, P., & Cliff, D., "Genetic Convergence in a Species of Evolved Robot Control Architectures", Cognitive Science Research Paper, Serial No. CSRP 267, University of Sussex. (1993 - Jan).

[20] Holland, J. H. “Hierarchical descriptions of universal spaces and adaptive systems”, Technical Report ORA Projects 01252 and 08226) Ann Arbor: University of Michigan, Department of Computer and Communication Sciences. (1968)

[21] Husbands, P., Harvey, I., & Cliff, D. "Analysing Recurrent Dynamical Networks Evolved for Robot Control". CSRP 265, University of Sussex. (1993 - Jan).

[22] Llinas, R. R., “The intrinsic electophysiological properties of mammalian neurons: Insights into central nervous system function”, Science 242, 1654-1664, (1988).

[23] McCartney, R. "Masters Specifications Report", School of Electronic Engineering, Dublin City University. (1992).

[24] McCartney, R. "DCU INTRA Report", School of Electronic Engineering, Dublin City University. (1992).

[25] McCartney, R. "Development of a Framework using Simple Genetic Algorithms for the Automated Design of Heterogenous Neural Networks for Robot Control", Thesis submitted for the award of the degree of B.Eng, School of Electronic Engineering, Dublin City University. (1993-1994).

[26] McCartney, R. "An Experimental Framework for Evolutionary Robotics", School of Electronic Engineering, Dublin City University. (1992).

[27] MacGregor, R. J., “Neural and Brain Modelling”, Academic Press, (1987).

[28] Microtech Research Inc. "Paragon™ Software Products Manual". (1985).

[29] Minsky, M. & Papert, S. "Perceptrons", Cambridge, MA: MIT Press. (1969)

[30] Motorola. "MC68000 Educational Computer Board User's Manual", Motorola, 2ndedition. (1982).

[31] Rumelhart, D. E., & McClelland, J. L., Editors, “Parallel Distributed Processing, Volume 1: Foundations”, M.I.T. Press-Bradford Books, (1986).

[32] Selverston, A. I., “A consideration of invertebrate central pattern generators as computational data bases”, Neural Networks, 1(2):109-117, (1988).

[33] Wilcox, A."68000 Microcomputer Experiments Using the Motorola Educational Computer Board", Prentice-Hall Inc. (1991).

[34] Wilson, Stewart W., "The Genetic Algorithm and Simulated Evolution", C. Langton (Ed.), Artificial Life, SFI Studies in the science of Complexity, Addison-Wesley, (1988).

Appendix A Paragon C cross Compiler Configuration file.

```
chip    68000
*
****************              Starting address for code and stack
*                             Stack grows down. Heap grows up from top
*                             of code
*
base    $2300
*
****************              Initializes stack pointer (sp)
*
public  ????STACKTOP=$2000
*
****************              Loader options
*
list t,x
*
****************              Main entry point (must be loaded first)
*
load    dstart
*
**************************************************************************
*                                                                        *
*       User supplied routines start here (there can be lots and lots)   *
*                                                                        *
**************************************************************************
*
load    NETLESS
*
**************************************************************************
*                                                                        *
*       The rest should not vary from program to program                 *
*                                                                        *
**************************************************************************
*
*
****************              Character output routine
load    outchr
****************              Character input routine
load    inchrw
****************              Heap management
load    sbrk
****************              Return to Monitor via TRAP #14
load    exit
****************              Startup and Level 2 I/O
load    csys68k
****************              C run-time library
load    \mcc68k\mcc68k.lib
*
end
**************************************************************************
```

Appendix B1 Derivation of Formula used for Update of Neural Network Input Parameters.

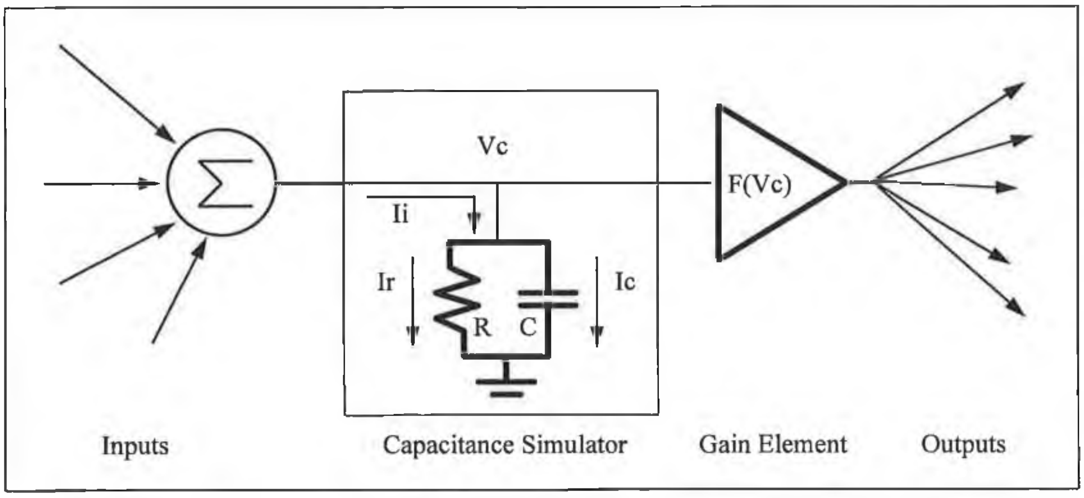


Figure B1: Neural Node used in Simulator and Simulation

Using Notation From Figure B:

$$I_i(t) = I_r(t) + I_c(t) \qquad (1)$$

$$I_r(t) = \frac{V_c(t)}{R} \qquad (2)$$

...by definition

also

$$I_c(t) = C\frac{dV_c(t)}{dt}$$

...by definition

Which can be approximated by:

$$I_c(t) \approx \frac{C \times \Delta V_c(t)}{\Delta t}$$

re-arranging we get

$$\Rightarrow \Delta V_c(t) = \frac{\Delta t \times I_c(t)}{C}$$

or alternately...

$$V_c(t) - V_c(t-1) = \frac{\Delta t \times I_c(t)}{C}$$

since

$$\Delta V_c(t) = V_c(t) - V_c(t-1).$$

$$\Rightarrow V_c(t) = V_c(t-1) + \frac{\Delta t \times I_c(t)}{C}$$

from (1) $\Rightarrow$

$$V_c(t) = V_c(t-1) + \frac{\Delta t \times [I_i(t) - I_r(t)]}{C}$$

from (2) $\Rightarrow$

$$V_c(t) = V_c(t-1) + \frac{\Delta t \times \left[I_i(t) - \frac{V_c(t)}{R} \right]}{C}$$

Rearranging we get:

$$C \times V_c(t) = C \times V_c(t-1) + \Delta t I_i(t) - \frac{\Delta t \times V_c(t)}{R}$$

and again to get

$$V_c(t)\left[C + \frac{\Delta t}{R}\right] = [C \times V_c(t-1)] + [\Delta t \times I(t)]$$

and finally:

$$\boxed{V_c(t) = \frac{[C \times V_c(t-1)] + [\Delta t \times I_i(t)]}{[C + \frac{\Delta t}{R}]}}$$

Appendix B2 Beers Formula for Update of Neural Network Input Parameters.

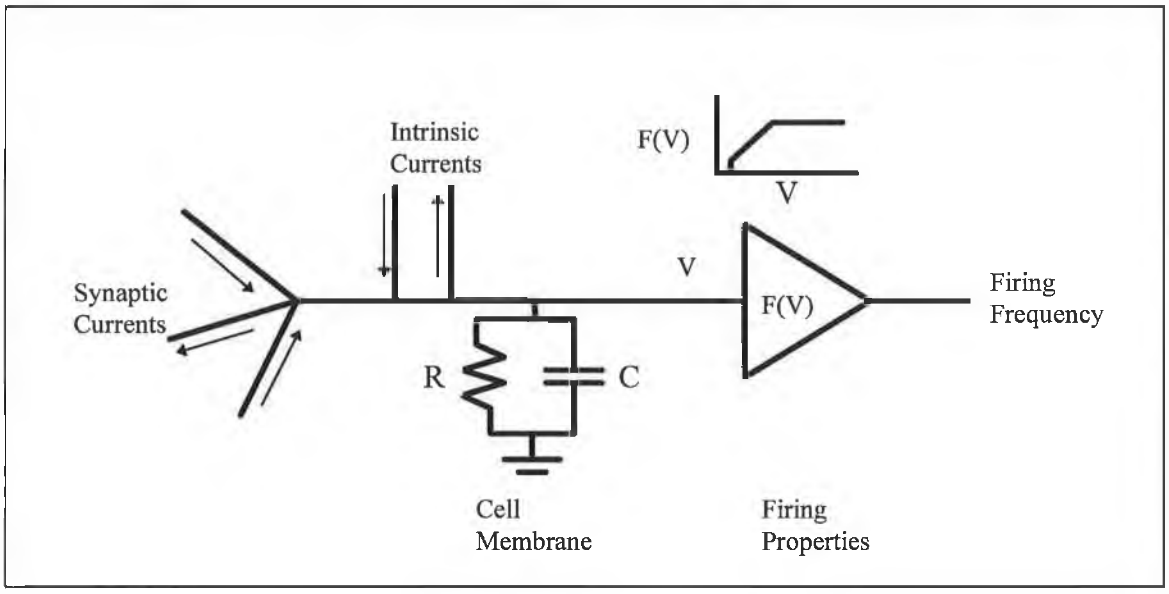


Figure B2 Neural Node structure implemented by Beer

$$\underbrace{C_N \frac{V_N(t)}{dt}}_{\text{Net Input Current}} = \underbrace{\sum_{M \in pre(N)} S_{M,N} F_M(V_M(t))}_{\text{Synaptic Currents}} + \underbrace{\sum_{L \in Intrinsic(N)} INT_L(t, V_N(t))}_{\text{Intrinsic Currents}} + \underbrace{EXT_N}_{\text{External Current}} - \underbrace{V_N(t)\,G_N}_{\text{Leak Current}}$$

where

C_N is the membrane capacitance of neuron N

$V_N(t)$ is the membrane potential of neuron N

$pre(N)$ is the set of neurons which form synapses on neuron N

$S_{M,N}$ is the strength of the connection from neuron N to neuron M

$F_M(V_M(t))$ is the firing frequency of neuron M

$Intrinsic(N)$ is the set of intrinsic currents of neuron N

$INT_L(t, V_N(t))$ is the magnitude of intrinsic current L,
which may be voltage and time dependant

EXT_N is the magnitude of external current injected into neuron N

G_N is the membrane conductance of neuron N

Reproduced from Beer[2][p 51]

Appendix C Random Number Generation Suite used in Simple Genetic Algorithm Program.

C.1 FILE: randutl.h

```
FILE: randutl.h

/*

ABSTRACT: Provides for sampling a random variable based on a
               variety of probability functions.

               This module is layered on msr88.

    Copyright (C) 1992  Barry McMullin.

    This is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 1, or any later version.

    This software is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this software, in the file LICENSE.DOC; if not,
    write to the Free Software Foundation, Inc., 675 Mass Ave,
    Cambridge, MA 02139, USA.

      Barry McMullin,
      School of Electronic Engineering,
      Dublin City University,
      Dublin 9,
      IRELAND.

      Telephone: +353 1 704 5432
      FAX:       +353 1 704 5508
      E-mail:    McMullinB@DCU.IE

*/

typedef int32 bernoulli_t;

extern bernoulli_t *setup_bernoulli(double p);
  /* Returns the appropriate structure to feed into get_bernoulli(), to
     make a sample of bernoulli random variable with parameter p.
     To free the memory allocated to the structure, use
     free_bernoulli();
     <p> must be in the range 0 to 1.
  */

extern void free_bernoulli(bernoulli_t **b);
  /* Frees the memory allocated for the structure <**b>. It
     leaves the pointer variable at the calling site <*b> as NULL. */

extern flag get_bernoulli(bernoulli_t *b);
  /* Returns TRUE according to a bernoulli probability function.
     <*b> should have been previously setup by setup_bernoulli();
  */
```

```
extern flag get_one_bernoulli(double p);
  /* This combines the functions of setup_bernoulli(),
     get_bernoulli(), and free_bernoulli().  It should be used
     whenever a *once off* bernoulli evaluation is required.  It
     returns TRUE with probability <p>.
     <p> must be in the range 0 to 1.
  */

typedef struct
{
   int32 n;
   int32 cutoff;
} uniform_t;


extern uniform_t *setup_uniform(int32 n);
  /* Returns the appropriate structure to feed into
     get_uniform(), to make a sample of a random number in the
     range 0 .. (<n>-1), with a uniform probability function. The
     slightly peculiar way of specifying the range (0 up to
     (<n>-1) instead of, say, 1 up to <n>) is deliberate: it
     means you can specify an array *size* and get back out a
     random *index*.

     To free the memory allocated to the structure, use
     free_uniform();

     <n> must be in the range 1 .. MSR88_RANGE. */

extern void free_uniform(uniform_t **u);
  /* Frees the memory allocated for the structure <**u>. It
     leaves the pointer variable at the calling site <*u> as NULL. */

extern int32 get_uniform(uniform_t *u);
  /* Returns a random number sampled according to a uniform
     probability function. <*u> should have been previously setup
     by setup_uniform();
  */

extern int32 get_one_uniform(int32 n);
  /* This combines the functions of setup_uniform(),
     get_uniform(), and free_uniform().  It should be used
     whenever a *once off* uniform evaluation is required.  It
     returns a random number in the range 0 .. (<n>-1), with a
     uniform probability function.

     <n> must be in the range 1 .. MSR88_RANGE. */


typedef struct
{
    int32 offset, range;
    int32 *table;
} cpf_t; /* Cumulative probability function type. */

extern flag valid_cpf(cpf_t *cpf);
  /* This returns TRUE iff <cpf> is a validly formatted
     cumulative probability function, as follows:

     <cpf->offset> is the minimum value of the random variable.
     There are no restrictions (other than those of the native
     type) on its value.
     <cpf->range> > 0, <= MSR88_RANGE. This is the total number of
     distinct values for the random variable.  The maximum value
     of the random variable will thus be
     <cpf->range + cpf->offset - 1>.

     <cpf->table> is a pointer to an array of values of the
     cumulative probability function, scaled to a maximum value
     of MSR88_RANGE.  This pointer must not be NULL. There must
     be exactly <cpf->range> entries in this array (the
     ``existance'' of these is checked only in the sense that
     there will be an attempted access to all of them).
     <cpf->table[0]> corresponds to a random variable value of
     <cpf->v_lo> etc.  No entries in the array may be less than
     zero or greater than MSR_RANGE.  The function must be
     monotonically increasing, and the final point should have
     the value MSR88_RANGE.

     */

extern void free_cpf(cpf_t **cpf);
  /* Frees the memory allocated for the structure <cpf>. It
     leaves the pointer variable at the calling site as NULL. */

extern int32 get_cpf(cpf_t *cpf);
  /* This returns a sample of a (pseudo) random variable with the
     probability function described by <cpf>. <cpf> must be a
     validly formatted probability function; in the interests of
     execution speed, *THIS IS NOT CHECKED* by get_cpf(). If in
     doubt, use valid_cpf() to check at the calling site. */



#define BINOMIAL_MAX ((int32)(100000L))
  /* This is the maximum n parameter for the binomial
     probability function supported here.  The value is somewhat
     arbitrary. It must be significantly less than MSR88_RANGE if
     the binomial probability function is to be "reasonably" well
     approximated. At the very best, we can attribute probability
     in units of about 1/MSR88_RANGE; if the required values are
     of this same order (or less) we will have a very poor
     approximation.  This is probably (;-) OK for those parts of
     the binomial range which contribute very little to the
     overall total, but would not be nice if it affected points
     in or around the mean.  Now, the probabilities here must be *at
     least* of the order 1/n (since n of them add up to 1) so if
     we keep n a good deal less than MSR88_RANGE we should be OK.
     But the details are still going to be application specific. */

extern cpf_t *setup_binomial_cpf(int32 n, double p);
  /* This sets up a cumulative probability function (cpf)
     for a binomial random variable with parameters <n> and
     <p>, and returns a pointer to a cpf_t data structure
     describing it. The cpf_t format is compatible with
     the function get_cpf(); the caller should not otherwise
     manipulate it. The structure is dynamically allocated; it
     can be deallocated by a call to free_cpf().

     <p> must be in the range 0 to 1.
     <n> must be in the range 1 to BINOMIAL_MAX. */
```

C.2 FILE: etc.h

```
     FILE: etc.h

     ABSTRACT:

     This is general purpose miscellaneous module, defining
     standard constants, and types, and a few handy functions etc.

    Copyright (C) 1992  Barry McMullin.

    This is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 1, or any later version.

    This software is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this software, in the file LICENSE.DOC; if not,
    write to the Free Software Foundation, Inc., 675 Mass Ave,
    Cambridge, MA 02139, USA.

      Barry McMullin,
      School of Electronic Engineering,
      Dublin City University,
      Dublin 9,
      IRELAND.

      Telephone: +353 1 704 5432
      FAX:       +353 1 704 5508
      E-mail:    McMullinB@DCU.IE

*/

/* Standard Constants: */

#define TRUE    1
#define FALSE   0

#define SUCCEED 0
#define FAIL    (-1)

#define EOL     '\n'
#define EOS     '\0'
#define BEL   '\007'


/* Standard Types: */

typedef          char   flag;           /* >=  1 bit, used as boolean  */

typedef unsigned long   unsigned32;     /* >= 32 bits, unsigned */
typedef          long   int32;          /* >= 32 bits, signed */

#define UNSIGNED32_MAX (0xFFFFFFFFL)
#define INT32_MAX      (0X7FFFFFFFL)
#define INT32_MIN      (- INT32_MAX)    /* Omit "negative zero"... */
```

```
/* Standard (default) string length: */

#define DEFAULT_STR_LEN (128)


/* Scope control Pseudo-keywords - explicitly specify *all*
   external objects as "public", "private" or "extern".                    */

#define public                      /* public is C default scope            */
#define private static              /* static *really* means private        */


#define strequ(s1, s2) (strcmp(s1, s2) == 0)
  /* String equality macro: */

extern flag valid_fraction(double p);
  /* Returns TRUE if 0 <= p <= 1. Useful for checking that
     probabilities *are*. */

extern flag fequ(double x, double y, double tol);
  /* Returns TRUE if the *fractional* difference between x and y
     (referred to x) is less than tol. */

extern int32 fround(double x);
  /* Converts x from double to int32, with rounding rather than
     truncation. */

#ifdef __TURBOC__

#define getkey() (bioskey(0) & 0xff)

#endif

#ifdef __GNUC__

extern char *strlwr(char *s);
  /* Convert s to lower case only... */

extern char *strupr(char *s);
  /* Convert s to upper case only... */

public double fabs(double x);


void* malloc(unsigned);
void  free(void *);
public ldiv_t ldiv(long num, long denom);
/* These should already be in <stdlib.h>.
   I think their absence is a gcc bug. */

#endif
```

C.3 FILE: msr88.h

```
FILE: msr88.h

     ABSTRACT:

     This module provides an implementation of the "minimal standard"
     pseudo random number generator defined in:

        Park, S.K
        Miller, K.W.
        "Random Number Generators: Good ones are hard to find."
        CACM, Oct 88, Vol. 31, No. 10, pp. 1192-1201.

     The name "msr88", however, is my own invention.

    Copyright (C) 1992  Barry McMullin.

    This is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 1, or any later version.

    This software is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this software, in the file LICENSE.DOC; if not,
    write to the Free Software Foundation, Inc., 675 Mass Ave,
    Cambridge, MA 02139, USA.

      Barry McMullin,
      School of Electronic Engineering,
      Dublin City University,
      Dublin 9,
      IRELAND.

      Telephone: +353 1 704 5432
      FAX:       +353 1 704 5508
      E-mail:    McMullinB@DCU.IE

*/

#define MSR88_RANGE ((int32)(2147483646L)) /* 2^31 - 2 */
  /* This is the total number of distinct values which the
     generator spans. */

extern void set_count_msr88(int32 newcount);
  /* Initialise random number generator with given count.
     This is used to check if the generator rolls over.
     This *must* be between 0 and (MSR88_RANGE - 1),
     inclusive. */

extern void set_seed_msr88(int32 newseed);
  /* Initialise random number generator with given seed.
     This *must* be between 1 and MSR88_RANGE, inclusive.

     Note the following "standard" values:

                 1    "Start" of get_msr88() cycle (default seed).
     1 043 618 065    Element #     10 000 (0.0005%) of get_msr88() cycle.
```

```
      1 768 507 984     Element # 10 000 000 (0.5%)     of get_msr88() cycle.
      1 209 575 029     Element #100 000 000 (5%)       of get_msr88() cycle.

    */

extern int32 get_count_msr88(void);
  /* Return current count value. */

extern int32 get_seed_msr88(void);
  /* Return current seed value. */

extern int32 get_msr88(void);
  /* Return pseudo random value.

     The (pseudo) probability function is uniform over the range
     0 .. (MSR88_RANGE-1), inclusive. This is slightly different
     from the definition given by Park & Millar: their generator
     returned a value between 1 and MSR88_RANGE.  My version
     (derived simply by decrementing their's by one) is more
     convenient for certain applications. */
```

C.4 FILE: panic.h

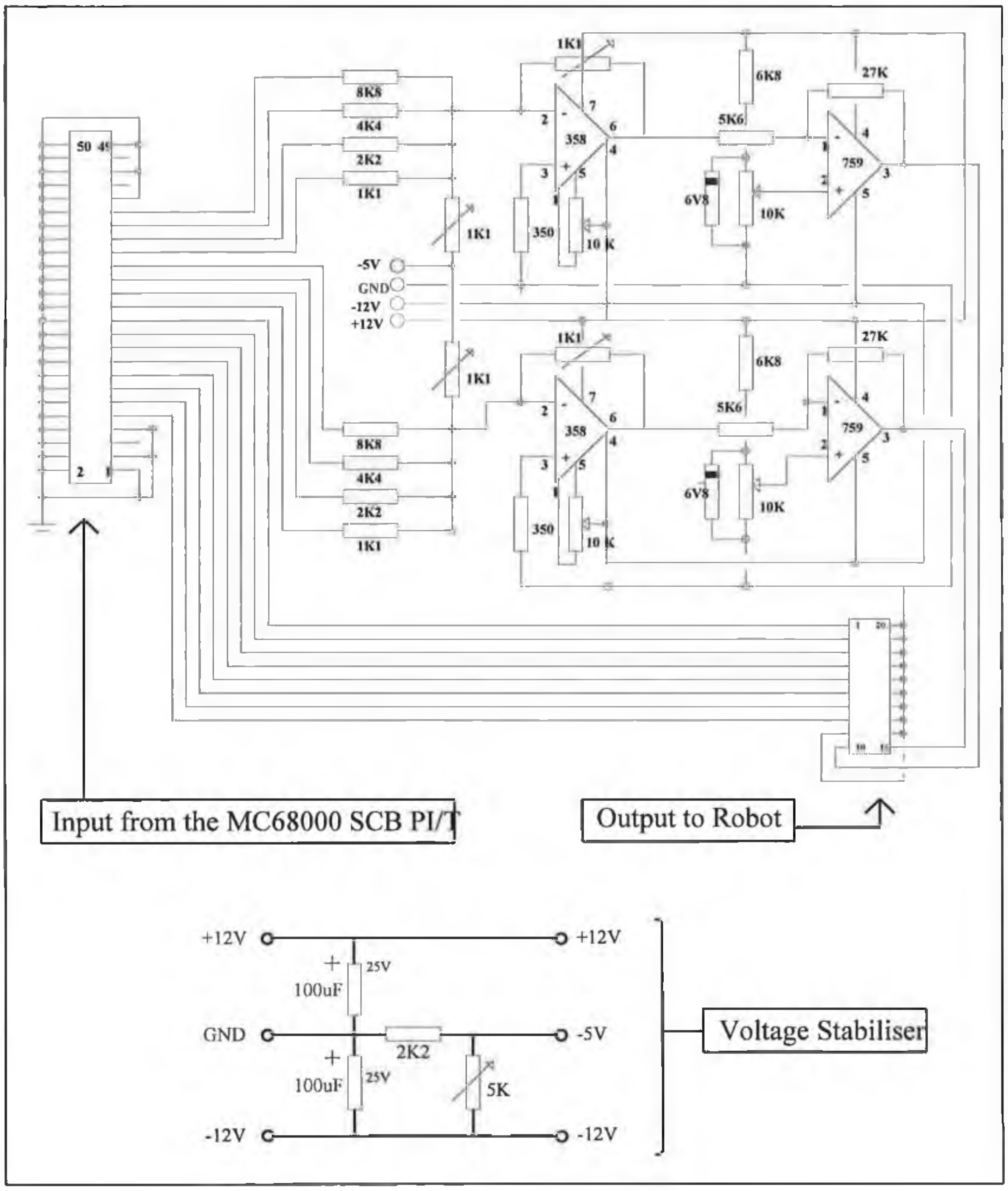
```
FILE: panic.h

     ABSTRACT:

     This file provides external declarations for
     a number of general purpose error reporting
     functions (via stderr).

    Copyright (C) 1992  Barry McMullin.

    This is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 1, or any later version.

    This software is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this software, in the file LICENSE.DOC; if not,
    write to the Free Software Foundation, Inc., 675 Mass Ave,
    Cambridge, MA 02139, USA.

      Barry McMullin,
      School of Electronic Engineering,
      Dublin City University,
      Dublin 9,
      IRELAND.

      Telephone: +353 1 704 5432
      FAX:       +353 1 704 5508
      E-mail:    McMullinB@DCU.IE

*/

extern char *process_name;

extern void warn(char *s);
/* General purpose WARNING. */

extern void panic(char *s);
/* General purpose PANIC. */

extern void syswarn(char *s);
/* WARNING on system call failure... */

extern void syspanic(char *s);
/* PANIC on system call failure... */
```

C.5 File: RANDOMS.C

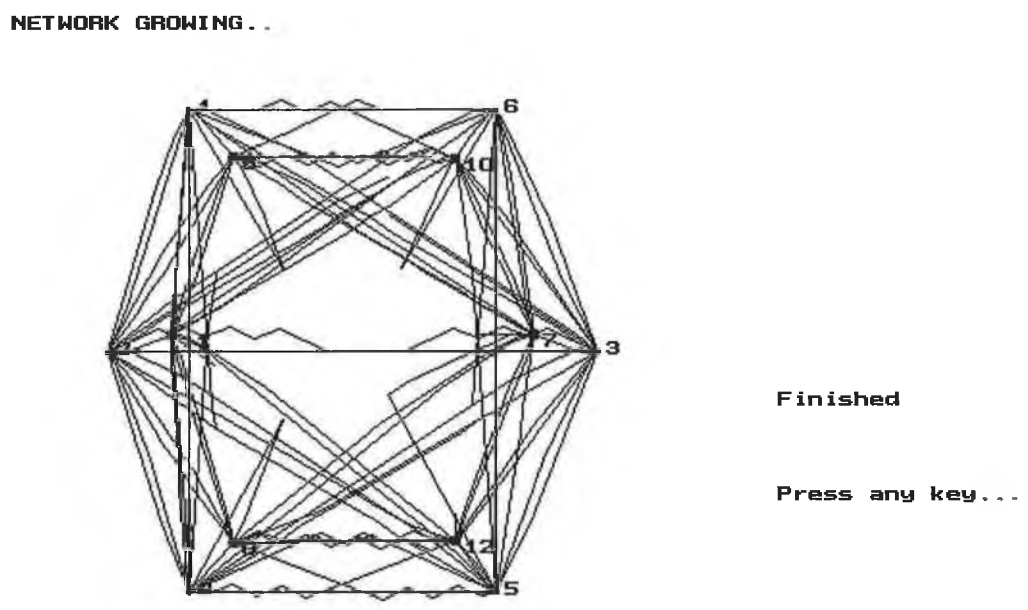
```
        File: RANDOMS.C
/*
        Program calls to the probability functions as used by the SGA
        crossover program.
*/

flag flip(double prob)
{
/* Function which simulates a coin toss with weighting
   determined by the probability figure passed in as a parameter.
*/
        return get_one_bernoulli(prob);
}

double gen_float(void)
{
/* Function which returns a value between 0 and 1.
*/
        return((double)(get_msr88())/(double)(MSR88_RANGE-1));
}

int get_cross_point(int range)
{
/* Function uses the random number generator function
   get_one_uniform() to choose a random crossing point for the
   CROSSOVER function.
*/
        return(get_one_uniform(range));
}


void getseed(void)
{
/* Function which gets the seed value for the random number
   generator from a file which is created by the program to save
   the seed value from the previous run of the genetic algorithm.
*/

        FILE          *seedfile;
        public int32 newseed;

        if ((seedfile = fopen("c:\\auto\\data\\seeddata.fil", "r+")) == NULL)
                {
                clrscr();
                fprintf(stderr, "Cannot open SEEDDATA.FIL for writing.\n");
                exit(1);
                }
                /* SEEDDATA.FIL CONTAINS THE RANDOM NUMBER GENERATOR SEED */

        fseek(seedfile,SEEK_SET, 0);
        fscanf(seedfile,"%Nli",&newseed);
        set_seed_msr88(newseed);
        fclose(seedfile);
}

Function which stores the current value of the random number
generators Seed.

void saveseed(void)
{
```

```
/* Function which stores the current value of the random number
   generators Seed.
*/

FILE         *seedfile;
public int32 oldseed;

       if ((seedfile = fopen("c:\\auto\\data\\SEEDDATA.FIL", "w+")) == NULL)
              {
              clrscr();
              fprintf(stderr, "Cannot open SEEDDATA.FIL for writing.\n");
              exit(1);
              }
              /* SEEDDATA.FIL CONTAINS THE RANDOM NUMBER GENERATOR SEED */

       fseek(seedfile,SEEK_SET, 0);

       oldseed = (int32)get_seed_msr88();
       fprintf(seedfile,"%Nli",oldseed);

       fclose(seedfile);
}
```

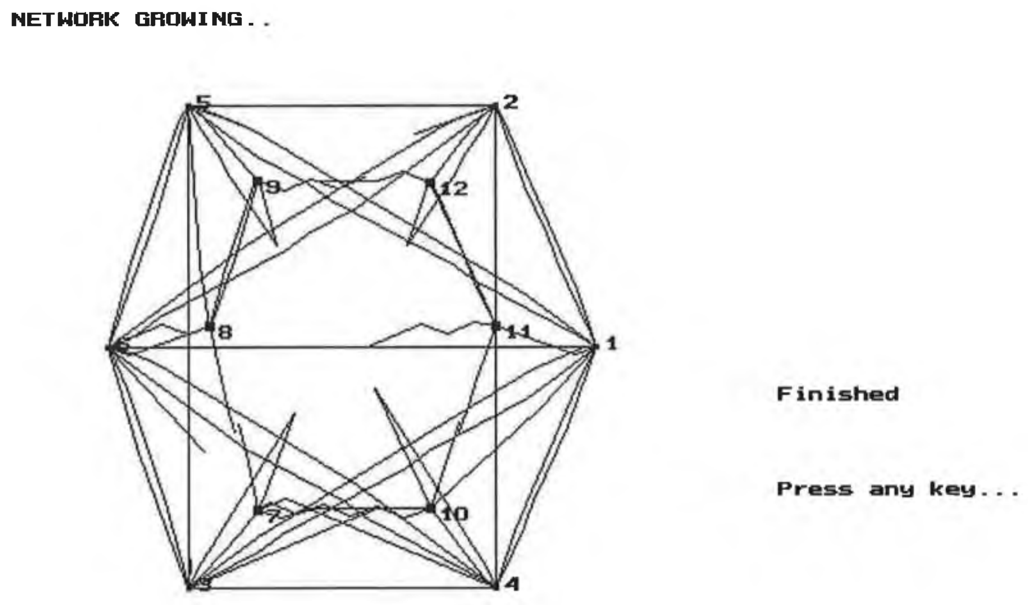
Appendix D Interface from MC68000 ECB PI/T to Robot



Appendix E Example of Networks Grown Using PCDEVEL

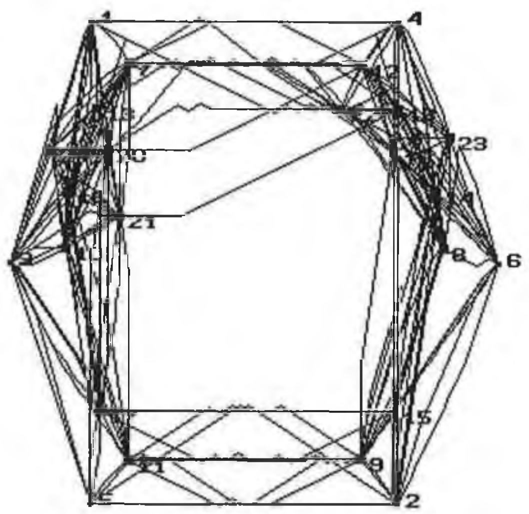


```
NETWORK GROWING..

                                              Finished

                                              Press any key...
```



```
NETWORK GROWING..

Finished

Press any key...
```

```
NETWORK GROWING..
```



```
Finished


Press any key...
```

Appendix F1 Network Development Code

```
/*      DEVEL.C
        THIS IS THE C PROGRAM WHICH CONTROLS THE DEVELOPMENT OF THE
        NEURAL NETWORK PHENOTYPE FROM THE BIT STRING GENOTYPE. TO
        OPERATE IT USES FUNCTIONS WITHIN THE C PROGRAMS:

        GETDATA.C       GRO_LINK.C      GRO_NODE.C      FORMS19.C

        DESCRIPTIONS OF THE FUNCTIONALITY OF THESE PROGRAMS IS
        GIVEN IN THEIR OWN CODES.

        THIS PROGRAM DEVEL.C IS USED TO CONTROL THE DEVELOPMENT OF THE
        NEURAL NETWORK TO OPERATE ON THE DEDICATED CONTROL BOARD.

        IT SEARCHES FOR THE TURBO C GRAPHICS FILES IN THE DIRECTORYS:

                D:\AUTO\
                C:\AUTO\

        THIS MUST BE SET UP BEFORE THE PROGRAM WILL BE COMPILED.*/
/*   NOTES : ....

        @       THE INPUT AND OUTPUT NODES ARE NOT DIRECTLY CONNECTED.

        @       THE OUTPUT NODES DO NOT GROW LINKS TO OTHER NODES :
                        THEY ARE PURELY RECEPTORS.*/

#define MEMORY_CHECK 0

#include "devel.h"
                /* STANDARD INCLUDE FILE FOR ALL DEVEL PROJECT FILES    */

#include <conio.h>
#include <ctype.h>
#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
extern unsigned _stklen;

/****************************************************************************

        STALL: FUNCTION TO CHECK GRAPHICS

****************************************************************************/

void graphics_check(int errorcode)
{
if (errorcode != grOk)
        {
        closegraph();
        printf("GRAPHICS ERROR : %s \n",grapherrormsg(errorcode));
        printf("\n\n\Any Key to Exit !!");
        exit(1);
        }
}
/****************************************************************************

        STALL: FUNCTION TO INITIALISE GLOBALS

****************************************************************************/

void initialize_globals(void)
{
int i = 0;
int j = 0;
```

```
 for(i = 0 ; i < MAX_NODES+5 ; i++)
        {
        for(j = 0 ; j < 3+1 ; j++)
               {
               nodes[i][j]=0;
               }
        }
               /* CONTAINS POSITION ON GROWTH GRID OF EACH NODE */

 for(i = 0 ; i < MAX_NODES+5 ; i++)
        {
        for(j = 0 ; j < 2+1 ; j++)
               {
               links[i][j]=0;
               }
        }

               /* STORES INFORMATION ON THE CURRENT LOCATION OF THE 'HEAD' OF */
               /* EACH LINK GROWING BETWEEN NODES */
 for(i = 0 ; i < 5+(MAX_NODES*MAX_NO_OP) ; i++)
        {
        for(j = 0 ; j < 2+1 ; j++)
               {
               conns[i][j]=0;
               }
        }
               /* STORES INFORMATION OF WHICH NODE IS CONNECTED TO WHICH AFTER  */
               /* CONNECTIONS BETWEEN NODES HAVE BEEN COMPLETED*/
 for(i = 0 ; i < MAX_NODES+5 ; i++)
        {
        for(j = 0 ; j < MAX_NO_OP+1 ; j++)
               {
               wghts[i][j]=0;
               }
        }

               /* STORES INFORMATION ON THE WEIGHT OF CONNECTIONS BETWEEN */
               /* CONNECTED NODES */
num_nodes= NUM_MOTORS + NUM_SENSORS;
               /* NUMBER OF NODES IN NETWORK. */
               /*      (INITIALISED TO THE NUMBER OF SENSORY MOTOR NODES */

num_conns=0;
               /* NUMBER OF CONNECTIONS MADE */

gen_ratio=0;
               /* RATIO OF GENERATOR NODES TO REACTIVE NODES  */

}
/***************************************************************************

        STALL: FUNCTION TO TERMINATE PROGRAM GRACEFULLY IN THE EVENT OF ERRORS
               OCCURING.

***************************************************************************/

void stall(void)
{
 closegraph();
               /* CLOSEDOWN GRAPHICS */

 exit(0);
               /* EXIT PROGRAM */
}

/***************************************************************************

        SET_GRAPHICS: FUNCTION TO INITIALISE GRAPHICS.

***************************************************************************/

void set_graphics(void)
```

```
{
        int gdriver = DETECT, gmode, errorcode;
                /* REQUEST AUTO DETECTION */

#if DDRIVE

 initgraph(&gdriver, &gmode, "D:\\AUTO");
                /* INITIALISE GRAPHICS MODE */
#else
 initgraph(&gdriver, &gmode, "C:\\AUTO");
                /* INITIALISE GRAPHICS MODE */

#endif

 errorcode = graphresult();
                /* READ RESULT OF INITIALISATION  */

 if (errorcode != grOk)
                /* AN ERROR OCCURED */

        {
        printf("Graphics error: %s\n", grapherrormsg(errorcode));
        printf("Press any key to halt:");
        exit(1);
                /* RETURN WITH ERROR CODE*/
        }

 setfillstyle(0,0);
                /* SET THE FILL PATTERN AND COLOUR */
        graphics_check(graphresult());

 cleardevice();
                /* CLEAR SCREEN AND RETURN 'CURSOR' POSITION TO (0,0)*/
        graphics_check(graphresult());

 setcolor(WHITE);
                /* SET FOREGROUND COLOUR */

}

/*****************************************************************************

        SETUP_IO:       FUNCTION TO INITIALISE I/O POSITIONS ON THE GROWTH GRID.

*****************************************************************************/

void setup_io(int *io_posit)
{
        int     x = 0;
        int     y = 1;
        int     iter=0;
        int     x_co_ord=0;
        int     y_co_ord=0;
        char msg[5];

 nodes[io_posit[0]-1][x]  = 50;        nodes[io_posit[0]-1][y]  = 50;
 nodes[io_posit[1]-1][x]  = 178;    nodes[io_posit[1]-1][y]  = 50;
 nodes[io_posit[2]-1][x]  = 306;    nodes[io_posit[2]-1][y]  = 50;
 nodes[io_posit[3]-1][x]  = 306;    nodes[io_posit[3]-1][y]  = 178;
 nodes[io_posit[4]-1][x]  = 306;    nodes[io_posit[4]-1][y]  = 306;
 nodes[io_posit[5]-1][x]  = 178;    nodes[io_posit[5]-1][y]  = 306;
 nodes[io_posit[6]-1][x]  = 50;     nodes[io_posit[6]-1][y]  = 306;
 nodes[io_posit[7]-1][x]  = 50;     nodes[io_posit[7]-1][y]  = 178;
                /* STORE POSITIONS OF THE SIX SENSOR AND 2 MOTOR NODES*/

 links[io_posit[0]-1][x]  = 50;     links[io_posit[0]-1][y]  = 50;
 links[io_posit[1]-1][x]  = 178;    links[io_posit[1]-1][y]  = 50;
 links[io_posit[2]-1][x]  = 306;    links[io_posit[2]-1][y]  = 50;
 links[io_posit[3]-1][x]  = 306;    links[io_posit[3]-1][y]  = 178;
 links[io_posit[4]-1][x]  = 306;    links[io_posit[4]-1][y]  = 306;
 links[io_posit[5]-1][x]  = 178;    links[io_posit[5]-1][y]  = 306;
 links[io_posit[6]-1][x]  = 50;     links[io_posit[6]-1][y]  = 306;
 links[io_posit[7]-1][x] = 50;      links[io_posit[7]-1][y] = 178;
              /* STORE POSITIONS OF THE HEAD OF THE FIRST LINK BEING GROWN BY*/
              /* EACH OF THE EIGHT EXISTING NODES WHICH IS THE SAME AS THE */
              /* LOCATION OF THE NODE THEMSELVES SINCE NO GROWTH HAS OCCURRED */

#if SHOW_GROWTH

 for (iter=0;iter<num_nodes;iter++)
        {
        x_co_ord = nodes[iter][x];
        y_co_ord = nodes[iter][y];
              /* GET LOCATION OF NODE FROM STORAGE ARRAY*/

        putpixel(x_co_ord,y_co_ord,WHITE);
              /* PLACE SINGLE POINT AT CENTRE OF NODE'S POSITION*/
        graphics_check(graphresult());
        rectangle(x_co_ord-1,y_co_ord-1,x_co_ord+1,y_co_ord+1);
              /* SURROUND THE POINT WITH A WHITE CIRCLE*/
        graphics_check(graphresult());
        sprintf(msg,"%d",iter);
        outtextxy(x_co_ord+5,y_co_ord-5,msg);
              /* PRINT NUMBER OF NODE BESIDE NODES LOCATION ON SCREEN*/
        graphics_check(graphresult());

        }

#endif /* SHOW_GROWTH */

}
/***************************************************************************

        SIM_GROWTH: MAIN CONTROLLING FUNCTION FOR THE DEVELOPMENT OF THE
              NEURAL NETWORKS BASED ON THE DECODED GENETIC PARAMETERS.

***************************************************************************/

void sim_growth(int lnk_grt,int div_rt,int div_ds,int *inbt,int num_cycles,
              int gene_num)
{
        int     count=0;
        int     iter=0;
        char type='a';
        char msg[50];

#if SHOW_GROWTH

        sprintf(msg,"NETWORK %d GROWING",gene_num);
        outtextxy(1,1,msg);
              /* ON SCREEN USER INFORMATION*/
              graphics_check(graphresult());
#endif /* SHOW_GROWTH */

grow_nodes(div_ds);

while (iter < num_cycles)
              /* LOOP ITERATES THE NUMBER OF TIMES DECODED FROM THE GENOTYPE*/

        {
        setfillstyle(EMPTY_FILL,GREEN);
              graphics_check(graphresult());

        bar(496,65,536,80);
              graphics_check(graphresult());

        setcolor(WHITE);
              graphics_check(graphresult());

        sprintf(msg,"Number of Cycles : %d",iter);
        outtextxy(356,70,msg);
              graphics_check(graphresult());

        bar(496,85,536,100);
```

```
                graphics_check(graphresult());

        setcolor(WHITE);
        graphics_check(graphresult());

        if( (sprintf(msg,"Number of Conns. : %d",num_conns))==EOF)
                {
                printf("ERROR !!!!!!!!!");
                exit(1);
                }

        outtextxy(356,90,msg);
                graphics_check(graphresult());
        if (num_nodes >(NUM_SENSORS + NUM_MOTORS))
                {
                grow_links(lnk_grt,inbt);

                        /* CALL TO FUNCTION WHICH CONTROLS THE GROWTH OF THE LINKS  */
                        /* WHICH JOIN THE INDIVIDUAL NODES */
                }
        if ((count>=div_rt)&&(div_rt!=0))
                /* IF TIME FOR NODAL GROWTH ASSUMING PARAMETER DOES NOT EQUAL 0*/

                {
                if (num_nodes>(MAX_NODES-(NUM_SENSORS+NUM_MOTORS)))
                        break;
                                /* BREAK IF NUMBER OF NODES WILL EXCEED THE MAXIMUM */
                                /* NUMBER OF ALLOWED NODES AFTER DIVISION. THE */
                                /* CONDITION ASSUMES WORST CASE IN THAT ALL EIGHT OF */
                                /* THE NODES IN THE PREVIOUS GENERATION WILL DIVIDE.*/
                else
                        grow_nodes(div_ds);
                                /* FUNCTION TO CONTROL NODAL GROWTH*/

#if SHOW_GROWTH

                setfillstyle(EMPTY_FILL,GREEN);
                        graphics_check(graphresult());

                bar(496,45,526,60);
                        graphics_check(graphresult());

                setcolor(WHITE);
                        graphics_check(graphresult());

                sprintf(msg,"Number of Nodes  : %d",num_nodes);
                outtextxy(356,50,msg);
                        /* UPDATE ONSCREEN FIGURE FOR NUMBER OF NODES IN NETWORK*/
                        graphics_check(graphresult());

#endif /* SHOW_GROWTH */

                count = 0;
                        /* RESET COUNT VARIABLE WHICH CHECKS FOR NODAL DIVISION */

                }
        count++;
        iter++;
        if(kbhit())
                {
                type = tolower(getch());
                if(type =='q')
                        stall();
                                /* ALLOWS USER TO QUIT AT ANY TIME BY PRESSING 'q'*/

                }
        }

  free(inbt);
                /* MEMORY MANAGEMENT */

 }
```

```
/***************************************************************************

        GET_INPUT_NUMBER:

***************************************************************************/
int get_input_number(void)
{
        int  gene_num = 0;
        int  input    = 0;

 clrscr();
 textcolor(WHITE);
 cprintf("\r\n\n\n\n\n\t\t\t\t  Not Enough Input Arguments\r\n");
 cprintf("\r\n\t\t\t\t  Call From Command Line With :");
 textcolor(LIGHTBLUE);
 cprintf("\r\n\r\n\t\t\t\t\t\t DEVEL X\r\n");
 textcolor(WHITE);
 cprintf("\r\n\t\t\t     Where X is an Integer Between 1 & 20 Inc.\r\n");
 cprintf("\r\n\t   Enter Chromosome Number or '0' to quit : ");

 scanf("%d",&input);

 if (input == 0 )
        stall();
                /* QUIT OPTION CHECK*/

 else
        {
        gene_num=input;

        }

 return(gene_num);
}

/***************************************************************************

        GRO_GENE:      CONTROLLING FUNCTION TO DECODE THE GENES.

***************************************************************************/

int gro_gene(int gene[GEN_LNGTH],int gene_num)
{
        int     *inhibit=NULL;
        int     num_cycles=0;
        int     lnk_grate=0;
        int     div_rate=0;
        int     div_dist=0;
        int     *io_posit=NULL;
        char msg[60];

 io_posit   = get_io_posit(gene);
 lnk_grate  = get_lnk_grate(gene);
 div_rate   = get_div_rate(gene);
 div_dist   = get_div_dist(gene);
 inhibit    = get_inhibit(gene);
 gen_ratio  = get_ratio(gene);
 num_cycles = get_num_cycles(gene);
 num_outputs= get_num_outputs(gene);

                /* ALL FUNCTIONS DESCRIBED IN DETAIL IN GETDATA.C*/

 setup_io(io_posit);
                /* SETUP THE INITIAL POSITIONS OF THE SIX SENSOR AND TWO MOTOR*/
                /* NODES BASED ON THE DECODED io_posit GENETIC VARIABLE*/
 free(io_posit);
 free(gene);

                /* MEMORY MANAGEMENT*/
#if SHOW_GROWTH

        sprintf(msg,"Division Rate       : %d",div_rate);
```

```
                outtextxy(50,350,msg);
        sprintf(msg,"Division Distance  : %d",div_dist);
                outtextxy(50,365,msg);
        sprintf(msg,"Link Growth Rate   : %d",lnk_grate);
                outtextxy(50,380,msg);
        sprintf(msg,"Generator Ratio    : %d",gen_ratio);
                outtextxy(50,395,msg);
        sprintf(msg,"Number of Cycles   : %d",num_cycles);
                outtextxy(50,410,msg);
        sprintf(msg,"Number of Outputs  : %d",num_outputs);
                outtextxy(50,425,msg);
        sprintf(msg,"Inhibitory Outputs : %d %d %d %d %d %d %d %d %d %d",
                inhibit[0],inhibit[1],inhibit[2],inhibit[3],
                inhibit[4],inhibit[5],inhibit[6],inhibit[7],
                inhibit[8],inhibit[9]);
                outtextxy(50,440,msg);
        sprintf(msg,"Negative Connection : ");
                outtextxy(356,110,msg);
        sprintf(msg,"Positive Connection : ");
                outtextxy(356,130,msg);
        setcolor(BLUE);
        line(526,113,556,113);
        setcolor(GREEN);
        line(526,133,556,133);
                        /* OUTPUT OF THE VALUES OF THE DECODES GENETIC VARIABLES 'ON */
                        /* SCREEN' FOR THE USER */

#endif /* SHOW_GROWTH */

sim_growth(lnk_grate,div_rate,div_dist,inhibit,num_cycles,gene_num);
                /* CALL TO MAIL NETWORK GROWTH FUNCTION WITH ALL THE DECODED */
                /* GROWTH VARIABLES PASSED AS PARAMETERS. IT WAS OPTED TO CALL    */
                /* THE GROWTH FUNCTION FROM HERE RATHER THAN RETURN ALL THE */
                /* VARIABLES TO THE MAIN FUNCTION JUST TO PASS THEM INTO THE */
                /* GROWTH FUNCTION FROM THERE.*/

return(0) ;

}

/*****************************************************************************

        VALIDATE_INPUT: FUNCTION WHICH CHECKS IF THE NUMBER INPUTTED ON THE
                COMMAND LINE IS VALID.

*****************************************************************************/

int validate_input(int input_number)
{
        int gene_num=0;

 gene_num = input_number;
 while((gene_num<1)||(gene_num>20))
        {
        clrscr();
        printf("\r\n\n\n\n\n\t\tNumber must be between 1 and 20.");
        printf("\r\n\n\t\tEnter Chromosome Number or '0' to quit: ");
                /* IF INCORRECT RANGE THEN CONTINUE TO PROMPT USER FOR THE */
                /* NUMBER OF THE CHROMOSOME TO DEVELOP */

        scanf("%d",&gene_num);
        if (gene_num == 0)
                exit(1)
                ;

        fflush(stdin);
                /*  FLUSH INPUT STREAM IN CASE OF BAD INPUT*/

        }
return(gene_num);
}
/*****************************************************************************
```

```
                                        MAIN FUNCTION.

****************************************************************************/

void main(int argc, char *argv[] )

{

        int             flg =0;
        int             *gene=NULL;
        int             i=1;
        int             gene_num=0;
        char    msg[30];
        long int        j=0;
 _stklen = 50000U;

#if MEMORY_CHECK

        printf("%lu bytes available\n",(unsigned long) coreleft() );
        getch();
        fflush(stdin);

#endif /* MEMORY_CHECK */

 if (argc!=2)
        gene_num = get_input_number();
                /* IF INCORRECT INPUT, THEN PROMPT USER TO INPUT THE DESIRED*/
                /* CHROMOSOME NUMBER TO BE DEVELOPED.*/

 else
        gene_num = validate_input (atoi(argv[1]));
                /* IF NUMBER IS INPUT THEN VALIDATE THE FIGURE.*/

 initialize_globals();

 set_graphics();
                /* INITIALISE GRAPHICS */

 gene = get_gene(gene_num);
                /* GET INPUT DATA (CHROMOSOME) FROM FILE */

 flg=gro_gene(gene,gene_num);

 if(flg != 0 )
        {
        closegraph();
        exit(0);
        }

                /* CALL TO ROUTINE TO 'GROW' THE NETWORK USING THE INFORMATION */

#if SHOW_GROWTH

        bar(0,0,170,20);

 while(!kbhit())
        {
        setcolor(i);
        sprintf(msg,"Finished Growth of Network %d :",gene_num);
        outtextxy(360,190,msg);
        outtextxy(360,210,"Press any key to continue");
        if(i==15)
                i=1;
        else
                i++;
        for(j=0;j<110000L;j++);
                /* ALLOWS CORRECT TIMING FOR FLASHING PROMPT (ASSUMING 486SX PC) */
        }
                /* PROMPT USER AT END OF GROWTH */

#endif /* SHOW_GROWTH */
create_s19();
                /* CALL TO FUNCTION TO GENERATE A MOTOROLA S19 DATA FILE FOR */
                /* TRANSMISION TO THE DEDICATED CONTROL BOARD.*/

stall();
                /* SHUTDOWN GRAPHICS AND END PROGRAM */
}
```

```
#define SHOW_GROWTH 0
#define SHOW_NODE_0 0
#define DDRIVE 0

#define GEN_LNGTH   54
#define MAX_NODES   50
#define MAX_NO_OP   10
#define NUM_SENSORS 6
#define NUM_MOTORS  2

#define MOTOR_A 6
#define MOTOR_B 7

int nodes[MAX_NODES+5][3+1];
               /* CONTAINS POSITION ON GROWTH GRID OF EACH NODE */

int links[MAX_NODES+5][2+1];
               /* STORES INFORMATION ON THE CURRENT LOCATION OF THE 'HEAD' OF */
               /* EACH LINK GROWING BETWEEN NODES */

int conns[5+(MAX_NODES*MAX_NO_OP)][2+1];
               /* STORES INFORMATION OF WHICH NODE IS CONNECTED TO WHICH AFTER  */
               /* CONNECTIONS BETWEEN NODES HAVE BEEN COMPLETED*/

int wghts[MAX_NODES+5][MAX_NO_OP+1];
               /* STORES INFORMATION ON THE WEIGHT OF CONNECTIONS BETWEEN */
               /* CONNECTED NODES */

int num_nodes;
               /* NUMBER OF NODES IN NETWORK. */
               /*     (INITIALISED TO THE NUMBER OF SENSORY MOTOR NODES */

int num_conns;
               /* NUMBER OF CONNECTIONS MADE */

int gen_ratio;
               /* RATIO OF GENERATOR NODES TO REACTIVE NODES */

int num_outputs;

int check_4_previous(int check_node,int base_node);
int get_lnk_grate(int buf[GEN_LNGTH]);
int get_div_rate(int buf[GEN_LNGTH]);
int get_div_dist(int buf[GEN_LNGTH]);
int get_num_cycles(int buf[GEN_LNGTH]);
int get_distance(int,int,int,int);
int get_ratio(int buf[GEN_LNGTH]);
int gro_gene(int *gene,int gene_num);
int get_num_outputs(int buf[GEN_LNGTH]);

void graphics_check(int errorcode);
void grow_nodes(int div_ds);
int *get_io_posit(int buf[GEN_LNGTH]);
int *get_inhibit(int buf[GEN_LNGTH]);
int *scene3(int x1,int x2,int y1,int y2,int axon_grth);
int *scene2(int x1,int x2,int y1,int axon_grth);
int *scene1(int x1,int y1,int y2,int axon_grth);
int *find_near(int base_node,int option);
int *get_gene(int gene_num);
int *get_point(int grth_dis,int x1,int y1,int x2,int y2);
void grow_links(int ax_grt,int *inbt);
void set_graphics(void);
void sim_growth(int ax_grt,int div_rt,int div_ds,int *inbt,int num_cycles,
               int gene_num);
void create_s19(void);
void stall(void);
```

```
/*  FORMS19.C

    This function generates S19 code for download os network information
    to the MC68000 ECB. It operates in conjunction with the simulator
    software: DEVEL.C calls the program.*/

#include "devel.h"
               /* STANDARD HEADER FILE*/
#include <stdio.h>
#include <alloc.h>

#define info_vol MAX_NODES*MAX_NO_OP*2
               /* TOTAL VOLUME OCCUPIED BY MAXIMUM SIZED NETWORK*/

static int address=(0x0000);
               /* ADDRESS WHERE NETWORK INFORMATION IS STORED IN THE DEDICATED   */
               /* CONTROL BOARD'S MEMORY*/

static char GEN_ADD[5]="0000\0";
               /* ADDRESS OF THE GENERATOR_RATIO VARIABLE*/

static char NUM_NOD[5]="0000\0";
               /* ADDRESS OF THE NUM_NODES VARIABLE*/

static char s19[46]="00000000000000000000000000000000000000000000\n";
               /* GLOBAL VARIABLE USED TO PRODUCE LINES OF S19 FORMAT DOWNLOAD   */
               /* INFORMATION FOR THE CONTROL BOARD.*/

/*****************************************************************************

       ADJUST_FORM: FUNCTION TO CONVERT A SINGLE DECIMAL NUMBER (0->15) TO
               HEXADECIMAL FORMAT (0->F)

*****************************************************************************/

char adjust_form(int value)
{
       char hex_num='g';

 switch (value)
       {
       case 15: hex_num = 'F'; break;
       case 14: hex_num = 'E'; break;
       case 13: hex_num = 'D'; break;
       case 12: hex_num = 'C'; break;
       case 11: hex_num = 'B'; break;
       case 10: hex_num = 'A'; break;
       case 9:  hex_num = '9'; break;
       case 8:  hex_num = '8'; break;
       case 7:  hex_num = '7'; break;
       case 6:  hex_num = '6'; break;
       case 5:  hex_num = '5'; break;
       case 4:  hex_num = '4'; break;
       case 3:  hex_num = '3'; break;
       case 2:  hex_num = '2'; break;
       case 1:  hex_num = '1'; break;
       case 0:  hex_num = '0'; break;

       default: hex_num = '0'; break;
       }
 return(hex_num);
}

/*****************************************************************************

       CALCULATE_CHECKSUM: FUNCTION TO CALCULATE THE ERROR CHECKING CHECKSUM
               FIGURE WHICH IS APPENDED TO EACH LINE OF S19 FORMAT CODE

*****************************************************************************/

void calculate_checksum(void)
{
        char hex_array[9] = "00000000";

        int checksum = 0;              int iter=0;             int num=0;
        int remain=0;                  int positn=7;

        char charac='g';
 for (iter=0;iter<8;iter++)
        {
        hex_array[iter] = '0';
        }
               /* INITIALISE ARRAY*/

 for (iter=2;iter<42;iter++)
        {
        charac = s19[iter];
        switch(charac)
               {
               case 'F': num = 15; break;
               case 'E': num = 14; break;
               case 'D': num = 13; break;
               case 'C': num = 12; break;
               case 'B': num = 11; break;
               case 'A': num = 10; break;
               case '9': num = 9; break;
               case '8': num = 8; break;
               case '7': num = 7; break;
               case '6': num = 6; break;
               case '5': num = 5; break;
               case '4': num = 4; break;
               case '3': num = 3; break;
               case '2': num = 2; break;
               case '1': num = 1; break;
               case '0': num = 0; break;
               }

        if(iter%2)
               checksum += num;
        else
               checksum += num * 16;
                      /* TO CALCULATE THE CHECKSUM THE HEXADECIMAL NUMBERS BEING  */
                      /* SENT ARE TREATED AS PAIRS SO THE NUMBERS HAVE TO BE */
                      /* ADJUSTED ACCORDINGLY*/

        }

 checksum = (0xFFF) - checksum;
               /* THE CHECKSUM FIGURE IS THE TWOS COMPLEMENT OF THE CALCULATED  */
               /* FIGURE*/

 while (checksum > 0)
        {
        remain = checksum % 16;
        checksum = checksum / 16;
        charac = adjust_form(remain);
     hex_array[positn] = charac;
        positn--;
                      /* ONLY THE TWO LSB's OF THE HEXADECIMAL CHECKSUM FIGURE   */
                      /* ARE USED.*/

        }

 for(iter=6;iter<8;iter++)
        {
        s19[iter+36]=hex_array[iter];
                      /* APPEND THE CHECKSUM TO THE S19 LINE*/
        }
}

/**************************************************************************

        Keeps S19 line format correct as each data bit is converted
        from decimal to hexadecimal.

****************************************************************************/

void update_s19(char data[33])
{
        char array[9] = "00000000";

        int iter=0;             int remain=0;           int positn=7;
        int number=0;

        char hex_char='g';
 for (iter=0;iter<8;iter++)
        {
        array[iter] = '0';
        }

 s19[0]='S';
 s19[1]='2';
 s19[2]='1';
 s19[3]='4';
                /* FIRST FOUR CHARACTERS ARE COMMON TO ALL LINES OF S19*/

 number = address;
 while (number>0)
        {
        remain         = number % 16;
        number         = number / 16;
        hex_char       = adjust_form(remain);
     array[positn] = hex_char;
     positn--;
        }

 for(iter=4;iter<8;iter++)
        {
        s19[iter+2]=array[iter];
        }

 for (iter=0;iter<32;iter++)
        s19[iter+10] = data[iter];
                /* CONSTRUCT S19 LINE*/

 calculate_checksum();
                /* CALCULATE THE CHECKSUM*/

 address += 16;
                /* UPDATE THE ADDRESS*/
}
/****************************************************************************

   SCENEA: FUNCTION WHICH CREATES HEXADECIMAL FOR NEGATIVE NUMBERS

****************************************************************************/

char *scenea(int number)
{

        int number1;            int iter=0;             int remain=0;
        int positn=7;

        char *array = NULL;

        char hex_char='g';
 if ((array = (char *)malloc(sizeof(char)*50)) == NULL)
        {
        printf("Not enough memory to allocate division_pt\n");
        stall();
                /* TERMINATE PROGRAM GRACEFULLY*/
        }
 for(iter=0;iter<8;iter++)
        *(array+iter) = 'F';
                        /* INITIALISE ARRAY*/
 number *= -1;

 if(number<4095)
        number1 = (0x1000);

 if(number<240)
        number1 = (0x100);

 if(number<16)
        number1 = (0x10);

 number= number1 - number;

 while (number>0)
        {
        remain           = number % 16;
        number           = number / 16;
        hex_char         = adjust_form(remain);
        *(array+positn)  = hex_char;
    positn--;
        }

return(array);
}
/*****************************************************************************

        SCENEB: FUNCTION WHICH CREATES HEXADECIMAL FOR POSITIVE NUMBERS

*****************************************************************************/

char *sceneb(int number)
{
        int remain=0;           int positn=7;          int iter=0;

        char charac='g';

        char *array = NULL;
  if ((array = (char *)malloc(sizeof(char)*50)) == NULL)
        {
        printf("Not enough memory to allocate division_pt\n");
        stall();
               /* TERMINATE PROGRAM GRACEFULLY*/
        }

 for (iter=0;iter<8;iter++)
        {
        *(array+iter)  = '0';
                       /* INITIALISE ARRAY*/

        }

 while (number >0)
        {
        remain = number % 16;
        number = number / 16;
        charac = adjust_form(remain);
        *(array+positn) = charac;
    positn--;
        }

return(array);
}
/*****************************************************************************

        CHANGE_DATA_FORM: FUNCTION READS IN 4 PIECES OF DECIMAL DATA IN AN
               ARRAY AND CONVERTS THEM TO HEXADECIMAL FORMAT USING OTHER
               FUNCTIONS IN THE PROGRAM.

*****************************************************************************/

char *change_data_form(int data[4])
```

```
{
        int iter=0;             int itrb=0;             int dat=0;

        char *charac=NULL;

        char *hex_array=NULL;

if ((hex_array=(char *)malloc(sizeof(char)*33))==NULL)
        {
        printf("\n out of memory");
        stall();
                /* TERMINATE PROGRAM GRACEFULLY*/

        }

 for(iter=0;iter<4;iter++)
        {
        dat = data[iter];

        if (dat<0)
                charac=scenea(dat);
                                /* IF NEGATIVE NUMBER*/
        else
                charac=sceneb(dat);
                                /* IF POSITIVE NUMBER*/

        for (itrb=0;itrb<8;itrb++)
                {
                hex_array[iter*8+itrb] = charac[itrb];
                }
        free(charac);
                        /* MEMORY MANAGEMENT*/

        }

hex_array[32] = '\0';
                /* END OF STRING MARKER*/

return(hex_array);
                /* RETURN HEXADECIMAL ARRAY*/

}
/******************************************************************************

        FORM_INFO: FUNCTION WHICH TAKES THE ARRAYS CONTAINING THE INFORMATION
                ABOUT NODES AND LINKS AND WEIGHTS AND REFORMATS IT INTO A SINGLE
                ARRAY RECOGNISABLE BY THE SIMULATOR PROGRAM.

******************************************************************************/

int *form_info(void)
{
        FILE *stream=NULL;

        int *buffer=NULL;

        int c_record[MAX_NODES]={0};

        int iter=0;             int node=0;             int conn=0;
        int posi=0;             int weit=0;
buffer = (int *)calloc( (info_vol+MAX_NO_OP),sizeof(int));
   if (buffer==NULL)
       {
       printf("Allocation failed in FORMS19.C (int* form_info)");
         stall();
                        /* ALLLOW GRACEFUL TERMINATION OF THE PROGRAM*/

       }

 for(iter=0;iter<(MAX_NO_OP*(MAX_NODES+1)*2);iter++)
        {
        buffer[iter]=0;
        }
                        /* INITIALISE ARRAY*/

#if DDRIVE

   stream = fopen("d:\\auto\\data\\netlist.fil","w");
                        /* OPEN FILE FOR NETWORK LISTING*/

#else
   stream = fopen("c:\\auto\\data\\netlist.fil","w");
                        /* OPEN FILE FOR NETWORK LISTING*/

#endif /* DDRIVE */

 fprintf(stream,"\t%s\n","NODE    TO    WGT");
                        /* HEADER FOR FILE*/

 for(iter=0;iter<(MAX_NO_OP*MAX_NODES);iter++)
                        /* LOOP TILL ALL INFORMATION IS OUTPUT*/

        {
        node = conns[iter][0];
                        /* NODE */

        posi = c_record[node];
                        /* STORE FOR NUMBER OF RECORDED OUTPUTS*/
                        /* INITIALLY ALL NODES HAVE 0 OUTPUTS*/

        if ( posi < MAX_NO_OP)
                        /* IF STILL BELOW MAXIMUM ALLOWED OUTPUT NUMBER*/
                {
                conn = conns[iter][1];
                        /* BASE NODE IS CONNECTED TO NODE 'conn'*/

                weit = wghts[node][posi];
                        /* WEIGHT OF CONNECTION*/

                fprintf(stream,"\n\t %d       %d       %d",node,conn,weit);
                                /* OUTPUT TO NETWORK LISTING FILE*/

                buffer[posi+(MAX_NO_OP*2*node)]=conn;
                buffer[posi+(MAX_NO_OP*2*node)+MAX_NO_OP]=weit;
                                /* ORGANISE MAIN INFORMATION ARRAY*/

                c_record[node]++;
                                /* INCREMENT CONNECTION RECORD FOR EACH NODE*/

                }
        }
 return(buffer);
}
/****************************************************************************

        CREATE_S19: CONTROLLING FUNCTION TO CREATE S19 FORMAT CODE FROM
                NETWORK INFORMATION.

****************************************************************************/

void create_s19(void)

{
        FILE *f_point1=NULL;            FILE    *f_point2=NULL;

        int *data_buf=NULL;

        static int s_num=0;             static int iter=0;              static int offset=0;

        int data[4]={0};

        char *hex_data=NULL;
#if DDRIVE
```

```
        f_point1 = fopen("d:\\auto\\data\\NETW.ABS", "w");
                /* OPEN S19 FILE FOR WRITING ONLY*/

#else

        f_point1 = fopen("c:\\auto\\data\\NETW.ABS", "w");
                /* OPEN S19 FILE FOR WRITING ONLY*/

#endif /* DDRIVE */
fseek(f_point1, SEEK_SET, 0);
                /* RESET FILE POINTER*/

#if DDRIVE

        f_point2 = fopen("D:\\auto\\data\\net_var.fil","r");

#else

        f_point2 = fopen("C:\\auto\\data\\net_var.fil","r");

#endif /* DDRIVE */
fseek(f_point2, SEEK_SET, 0);

fscanf(f_point2,"%x%4s%4s",&address,&GEN_ADD,&NUM_NOD);

fclose(f_point2);
fprintf(f_point1,"%s\n","S00600004844521B");
                /* INITIAL SEGMENT COMMON TO ALL S19 FILES*/

 s19[0]='S';
 s19[1]='2';
 s19[2]='1';
 s19[3]='4';
 s19[4]='0';
 s19[5]='0';

 for(iter=6;iter<10;iter++)
        {
        s19[iter] = GEN_ADD[iter-6];
        }
 hex_data = sceneb(gen_ratio);
 for(iter=10;iter<18;iter++)
        {
        s19[iter] = hex_data[iter-10];
        }
 calculate_checksum();
 fprintf(f_point1,"%s",s19);
                /* OUTPUT THE GENERATOR RATIO FIGURE TO THE S19 FILE*/

 s19[0]='S';
 s19[1]='2';
 s19[2]='1';
 s19[3]='4';
 s19[4]='0';
 s19[5]='0';

 for(iter=6;iter<10;iter++)
        {
        s19[iter] = NUM_NOD[iter-6];
        }
 hex_data = sceneb(num_nodes);
 for(iter=10;iter<18;iter++)
        {
        s19[iter] = hex_data[iter-10];
        }
 calculate_checksum();
 fprintf(f_point1,"%s",s19);
                /* OUTPUT THE NUMBER OF NODES FIGURE TO THE S19 FILE*/

 data_buf = form_info();
                /* CALL TO FUNCTION TO ORGANISE THE STRUCTURE OF THE NETWORK */
                /* INFORMATION FOR SENSIBLE TRANSMISSION*/
```

```
for (iter=0;iter<MAX_NODES*20;iter++)
        {
        data[offset]= *(data_buf+iter);
                        /* GET NETWORK INFORMATION*/

        if(offset>=3)
                        /* WHEN 3 BITS OF INFORMATION HAVE BEEN OBTAINED*/

                {
                hex_data = change_data_form(data);
                        /* CONVERT TO HEXADECIMAL*/

                hex_data[32] = '\0';
                        /* TERMINATING STRING CHARACTER*/

                update_s19(hex_data);
                        /* FORM THE FULL S19 LINE OF INFORMATION*/

                fprintf(f_point1,"%s",s19);
                        /* OUTPUT TO THE s19 FILE */

                offset=0;
                s_num++;
                free(hex_data);
                        /* MEMORY MANAGEMENT AND VARIABLE RESET*/

                }
        else
                {
                offset++;
                }
        }

fprintf(f_point1,"\n");
fprintf(f_point1,"%s\n","S804002300D8");
                /* OUPUT THE TERMINATING S19 RECORD                                    */

free(data_buf);
                /* MEMORY MANAGEMENT*/

fclose(f_point1);
                /* CLOSE S19 DATA FILE*/

}
```

```
/*
        GET_DATA.C:

        PROGRAM WHICH READS IN THE GENOTYPES PRODUCED BY THE SGA OPERATION AND
        STORES IT IN USABLE FORM. IT IS USED BY THE DEVEL.C PROGRAM.

        IT ALSO CONTAINS THE CODE FOR DECODING THE GENOTYPES*/

#include "devel.h"
#include <stdio.h>
#include <alloc.h>
                /* STANDARD INCLUDE FILES */

/*****************************************************************************

        GET_IO_POSIT: FUNCTION TO DECODE THE RELEVANT SECTION OF BINARY CODED
                GENE AND CONVERT TO NUMERICAL FORMAT FOR THE RELATIVE POSITIONS OF
                THE  INPUT SENSORS AND OUTPUT MOTORS ON THE GROWTH GRID.

*****************************************************************************/

int *get_io_posit(int buf[GEN_LNGTH])
{
        int iter=0;
        int *io_coded;
                /* INPUT/OUTPUT SENSORS/MOTORS POSITION RELATIVITY IN GENOTYPE */
                /* CODED FORM.*/

        int *io_decoded;
                /* DECODED INPUT/OUTPUT SENSORS/MOTORS POSITION RELATIVITY  */

        int outer[NUM_MOTORS+NUM_SENSORS] = {1,2,3,4,5,6,7,8};
        int right=NUM_MOTORS+NUM_SENSORS-1;
                /* No. 1 SUBTRACTED FOR EASE OF ARRAY INDEX SELECTION*/

        int left=0;
        int pos_starter=0;
        int position=0;
if (( io_coded= malloc(460)) == NULL)
        {
        printf("Not enough memory to allocate buffer\n");
        stall();
                /*  TERMINATE PROGRAM IF OUT OF MEMORY */

        }

if ((io_decoded = malloc(460)) == NULL)
        {
        printf("Not enough memory to allocate buffer\n");
        stall();
                /*  TERMINATE PROGRAM IF OUT OF MEMORY */

        }

for (iter = 0;iter<12;iter++)
      {
       io_coded[iter] = buf[iter];
      }

 pos_starter += io_coded[8] * 8;
                /* MOST SIGNIFICANT BIT*/

 pos_starter += io_coded[9] * 4;
 pos_starter += io_coded[10] * 2;
 pos_starter += io_coded[11];
                /* BINARY TO DECIMAL CONVERSION*/

 position = pos_starter % (NUM_SENSORS + NUM_MOTORS);
                /* BASED ON DECODED STARTING POSITION BEGIN TO ACCESS THE ARRAY  */
                /* 'outer' TO DETERMINE ORDER OF SENSORS STARTING WITH           */
                /* 'outer[position]'.*/
 iter = 0;
 while (iter < (NUM_SENSORS + NUM_MOTORS) )
        {
        if (io_coded[position] == 1)
                {
                io_decoded[position] = outer[right];
                right--;
                }
                       /* TAKE NUMBER FROM THE RIGHT IF '1' IS READ*/

        else
                {
                io_decoded[position] = outer[left];
                left++;
                }
                       /* TAKE NUMBER FROM THE LEFT IF '0' IS READ*/

        position++;
                /* INCREMENT POSITION COUNTER*/

        if (position>= 8)
                position = 0;
                       /* WHEN END OF RELEVANT GENOTYPE SECTION RETURN TO THE       */
                       /* START OF SECTION*/

        iter++;
        }

 return(io_decoded);
}

/***************************************************************************

        GET_INHIBIT: FUNCTION TO DECODE THE RELEVANT SECTION OF BINARY CODED
                GENE AND STORE INFORMATION ON WHICH CONNECTIONS SHOULD BE
                POSITIVE OR NEGATIVE.

***************************************************************************/

int *get_inhibit(int buf[GEN_LNGTH])
{
        int iter=0;
        int count=0;
        int *inhibit = NULL;

 if ((inhibit = malloc(1000)) == NULL)
        {
        printf("Not enough memory to allocate buffer\n");
        stall();
                /*  TERMINATE PROGRAM IF OUT OF MEMORY */

        }
 for(iter=12;iter<=21;iter++)
        {
        *(inhibit+count) = 1;
        if (buf[iter]==0)
                {
                *(inhibit+count) = 0;
                }
        count++;
      }

return(inhibit);
}

/***************************************************************************

        GET_LNK_GRATE: FUNCTION TO CONVERT FROM 'BINARY' TO NUMERIC FORMAT
                FOR LINK GROWTH RATE.

***************************************************************************/
```

```
int get_lnk_grate(int buf[GEN_LNGTH])
{
       int lnk_grate = 0;

 lnk_grate += buf[22] * 8;
              /* MOST SIGNIFICANT BIT*/

 lnk_grate += buf[23] * 4;
 lnk_grate += buf[24] * 2;
 lnk_grate += buf[25];

 lnk_grate += 1;
              /* CONVERSION USING SIMPLE BINARY WEIGHTING                  */
              /* RANGE     0 -> 15 AND SCALING FROM 1 -> 16*/

 return(lnk_grate);
}
/*****************************************************************************

       GET_DIV_RATE: FUNCTION TO DECODE THE RELEVANT SECTION OF BINARY CODED
              GENE AND CONVERT TO NUMERICAL FORMAT FOR NODE DIVISION RATE.

*****************************************************************************/

int get_div_rate(int buf[GEN_LNGTH])
{
       int div_rate=0;

 div_rate += buf[26] * 32;
              /* MOST SIGNIFICANT BIT*/
 div_rate += buf[27] * 16;
 div_rate += buf[28] * 8;
 div_rate += buf[29] * 4;
 div_rate += buf[30] * 2;
 div_rate += buf[31];

 div_rate += 2;

              /* RANGE   2 -> 65 cycles/division */

return(div_rate);
}
/*****************************************************************************

       GET_NUM_CYCLES: FUNCTION TO DECODE THE RELEVANT SECTION OF BINARY CODED
              GENE AND CONVERT TO NUMERICAL FORMAT FOR THE NUMBER OF GROWTH
              CYCLES.

*****************************************************************************/

int get_num_cycles(int buf[GEN_LNGTH])
{
       int num_cycles = 0;

 num_cycles += buf[32] * 128;
              /* MOST SIGNIFICANT BIT*/

 num_cycles += buf[33] * 64;
 num_cycles += buf[34] * 32;
 num_cycles += buf[35] * 16;
 num_cycles += buf[36] * 8;
 num_cycles += buf[37] * 4;
 num_cycles += buf[38] * 2;
 num_cycles += buf[39];
 num_cycles += 20 ;

 if  (num_cycles > 255)
       num_cycles = 255;

              /* RANGE 20 -> 255*/

return(num_cycles);
```

```
}
/*****************************************************************************

        GET_RATIO: FUNCTION TO DECODE THE RELEVANT SECTION OF BINARY CODED
                GENE AND CONVERT TO NUMERICAL FORMAT FOR THE GENERATOR NODE
                RATIO.

*****************************************************************************/

int get_ratio(int buf[GEN_LNGTH])
{
        int ratio=0;

 ratio += buf[40] * 16;
                /* MOST SIGNIFICANT BIT*/

 ratio += buf[41] * 8;
 ratio += buf[42] * 4;
 ratio += buf[43] * 2;
 ratio += buf[44];

                /* RANGE 0 -> 31*/

 return(ratio);
}
/*****************************************************************************

        GET_DIV_DIST: FUNCTION TO DECODE THE RELEVANT SECTION OF BINARY CODED
                GENE AND CONVERT TO NUMERICAL FORMAT FOR THE NODE DIVISION
                DISTANCE.

*****************************************************************************/

int get_div_dist(int buf[GEN_LNGTH])
{
        int div_dist=0;

 div_dist += buf[45] * 32;
                /* MOST SIGNIFICANT BIT*/

 div_dist += buf[46] * 16;
 div_dist += buf[47] * 8;
 div_dist += buf[48] * 4;
 div_dist += buf[49] * 2;
 div_dist += buf[50];

 div_dist += 35;

                /* RANGE  35 -> 98 PIXELS                                    */

 return(div_dist);
}
/*****************************************************************************

        GET_NUM_OUT : FUNCTION TO DECODE THE RELEVANT SECTION OF BINARY CODED
                GENE AND CONVERT TO NUMERICAL FORMAT FOR THE NUMBER OF ALLOWED
                NODAL OUTPUTS.

*****************************************************************************/

int get_num_outputs(int buf[GEN_LNGTH])

{

 int outputs=0;

 outputs += buf[51] * 4;
                /* MOST SIGNIFICANT BIT*/

 outputs += buf[52] * 2;
 outputs += buf[53] * 1;
 outputs += 2;

               /* RANGE  2 -> 10 PIXELS                                    */

 return(outputs);
}
/**************************************************************************

       GET_GENE: FUNCTION TO GET THE GENETIC DATA FROM THE FILE POPDATA.FIL

**************************************************************************/

int *get_gene(int gene_num)
{
       FILE *f_point=NULL;
       int iter=0;
       int i=0;
       long offset=0;
       int *gene = NULL;

 if ((gene = (int *)malloc(sizeof(int)*60)) == NULL)
       {
       printf("Not enough memory to allocate buffer\n");
       stall();
               /*  TERMINATE PROGRAM IF OUT OF MEMORY */

       }
#if DDRIVE

   if ((f_point = fopen("D:\\auto\\data\\POPDATA.FIL", "r+")) == NULL)
       {
       fprintf(stderr, "Cannot open data file. C:\AUTO\data\POPDATA.FIL \n");
       stall();
               /* OPEN DATA FILE FOR READING ONLY AND EXIT PROGRAM GRACEFULLY */
               /* IF DATA IS NOT PRESENT*/

       }
#else

   if ((f_point = fopen("C:\\auto\\data\\POPDATA.FIL", "r+")) == NULL)
       {
       fprintf(stderr, "Cannot open data file. C:\AUTO\data\POPDATA.FIL \n");
       stall();
               /* OPEN DATA FILE FOR READING ONLY AND EXIT PROGRAM GRACEFULLY */
               /* IF DATA IS NOT PRESENT*/

       }

#endif /* DDRIVE */

 switch (gene_num)
       {
       case  1:       offset = 0L;   break;
       case  2:       offset = 55L;  break;
       case  3:       offset = 110L; break;
       case  4:       offset = 166L; break;
       case  5:       offset = 222L; break;
       case  6:       offset = 278L; break;
       case  7:       offset = 334L; break;
       case  8:       offset = 390L; break;
       case  9:       offset = 446L; break;
       case 10:       offset = 502L; break;
       case 11:       offset = 558L; break;
       case 12:       offset = 614L;  break;
       case 13:       offset = 670L; break;
       case 14:       offset = 726L; break;
       case 15:       offset = 782L; break;
       case 16:       offset = 838L; break;
       case 17:       offset = 894L; break;
       case 18:       offset = 950L; break;
       case 19:       offset = 1006L; break;
       case 20:       offset = 1062L; break;
```

```
        }

                /* POPDATA.FIL CONTAINS THE INFORMATION ON ALL THE MEMBERS OF    */
                /* OF THE GENETIC POPULATION SO TO ACCESS A PARTICULAR ONE IT IS */
                /* NECCESSARY TO USE AN OFFSET.*/

 fseek(f_point,offset,SEEK_SET);
                /* GO TO OFFSET POINT*/

 for (iter = 0; iter<GEN_LNGTH; iter++)
        {
        if (fscanf(f_point, "%li", &i))
                {
                gene[iter] = i;
                }
                        /* GET DATA AND STORE IN ARRAY*/

        else
                {
                fprintf(stderr, "Error reading from DATA.FIL !!\n");
                stall();
                        /* OR PRINT ERROR MESSAGE*/
                }
        }
 fclose(f_point);
 return(gene);
}
```

```
/*
        GROW_LINK.C
                THIS PROGRAM CONTROLS THE GROWTH OF LINKS BETWEEN NODES.*/
#include "devel.h"
#include <graphics.h>
#include <stdio.h>
#include <alloc.h>
#include <math.h>
#include <conio.h>

#if SHOW_NODE_0

   #include <stdlib.h>

#endif

                /* STANDARD INCLUDE FILES*/
/****************************************************************************

        CHECK_4_PREVIOUS: FUNCTION TO CHECK IF NODES ARE CONNECTED ALREADY.

****************************************************************************/

int check_4_previous(int check_node,int base_node)
{
        int iter = 0;
        int flag = 1;

for (iter=0;iter<num_conns;iter++)
        {
        if ( (conns[iter][0]==base_node) & (conns[iter][1]==check_node) )
                flag=0;
        }

return(flag);
}
/****************************************************************************

        GET_DISTANCE : FUNCTION TO FIND THE DISTANCE BETWEEN TWO POINTS
                               (x1,y1) and (x2,y2)

****************************************************************************/

int get_distance(int x1,int y1,int x2,int y2)
{
        int distance=0;

        double delta_x=0.0;                     double delta_y=0.0;
        double dx_squared=0.0;          double dy_squared=0.0;
        double to_be_rooted=0.0;

delta_x = x2-x1;
delta_y = y2-y1;

dx_squared = pow(delta_x,2);
dy_squared = pow(delta_y,2);

to_be_rooted = dx_squared + dy_squared;

                /* DISTANCE CALCULATED BY THE APPLICATION OF PYTHAGOROUS */

distance = (int) sqrt( to_be_rooted );

return(distance);
}

/****************************************************************************

        FIND_NEAR: FUNCTION TO FIND THE NEAREST TWO NODES RELATIVE TO A
                POSITION ON THE GROWTH GRID.

                THE OPTION IS USED TO DECIDE IF THE BASE POSITION SHOULD BE A
```

```
               NODE OR A LINK HEAD.

******************************************************************************/

int *find_near(int base_node,int option)
{
        int *near_2=NULL;

        int x1=0;       int y1=0;                       int x2=0;
        int y2=0;       int distance=0;         int iter=0;
        int flag=1;     int min_dist_1 = 1000; int min_dist_2 = 1000;
                                int start_figure=0;

 near_2 = (int *)calloc(20,sizeof(int));

 if(near_2 == NULL)
        {
        printf("Not enough memory to allocate near_2\n");
        stall();
                        /* TERMINATE PROGRAM GRACEFULLY*/
        }

 if (option == 0)
                /* OPTION ALLOWS THE FUNCTION TO BE USED TO CALCULATE THE */
                /* NEAREST NODE FOR EITHER THE NODE DIVISION ROUTINE OR THE LINK */
                /* HEAD SEARCH ROUTINE.*/
                /* 0 = LINK GROWTH.                     1 = NODE DIVISION*/

        {
        x1 = links[base_node][0];
        y1 = links[base_node][1];
        }
 else
        {
        x1 = nodes[base_node][0];
        y1 = nodes[base_node][1];
        }

 if ((base_node < NUM_SENSORS+NUM_MOTORS) && (option == 0))
        start_figure = NUM_SENSORS+NUM_MOTORS;
 else
        {
        if (option == 0)
                start_figure = NUM_SENSORS;
        else
                start_figure = 0;
        }
                /* prevent an i/o node connecting directly with another i/o node */
                /* and prevent internal input connections to an input node       */
 if (start_figure == num_nodes)
        {
        free(near_2);
        return(NULL);
        }

 for (iter=start_figure;iter<num_nodes;iter++)
        {
        if (option == 0)
                {
                flag = check_4_previous(iter,base_node);
                        /* CHECK FOR EXISTING CONNECTION */

                }

                        /* prevent link turning back towards base node */
        if ((iter !=base_node) && (flag != 0))
                {
                x2=nodes[iter][0];
                y2=nodes[iter][1];
                        /* SELECT NODE*/

                distance = get_distance(x1,y1,x2,y2);
```

```
                        /* CALCULATE DISTANCE BETWEEN POINTS*/

                if (distance<min_dist_2)
                        {
                        min_dist_2 = distance;
                        *(near_2+2) = iter;

                        if (min_dist_2 < min_dist_1)
                                {
                                *(near_2+2)  = *(near_2+1);
                                *(near_2+1)  = iter;
                                *(near_2+0)  = distance;

                                min_dist_2 = min_dist_1;
                                min_dist_1 = distance;
                                }
                        }
                }
        }
 return(near_2);
}

/****************************************************************************

        SCENE 1: FUNCTION WHICH DEALS WITH LINK GROWTH IN THE SITUATION
                WHERE THE TARGET NODE AND THE LINK HEAD ARE ALIGNED PARALLEL
                TO THE Y AXIS.

****************************************************************************/

int *scene1(int x1,int y1,int y2,int link_grth)
{
        int *to_point = NULL;

 to_point = (int *)calloc(10,sizeof(int));

 if(to_point == NULL)
        {
        printf("Not enough memory to alloc. scene1\n\n ANY KEY TO TERMINATE !");
        getch();
        stall();
                /* TERMINATE THE PROGRAM GRACEFULLY*/
        }

 to_point[0] = x1;
 if (y1<y2)
        {
        to_point[1] = y1 + link_grth;
        }
 else
        {
        to_point[1] = y1 - link_grth;
        }
 return(to_point);
}

/****************************************************************************

        SCENE2: FUNCTION WHICH DEALS WITH LINK GROWTH IN THE SITUATION
                WHERE THE TARGET NODE AND THE LINK HEAD ARE ALIGNED PARALLEL
                TO THE X AXIS.

****************************************************************************/

int *scene2(int x1,int x2,int y1,int link_grth)
{
        int *to_point=NULL;
 to_point = (int *)calloc(10,sizeof(int));

 if(to_point == NULL)
        {
        printf("Not enough memory to alloc. scene1\n\n ANY KEY TO TERMINATE !");
        getch();
        stall();
                /* TERMINATE THE PROGRAM GRACEFULLY*/
        }

 to_point[1] = y1;
 if (x1<x2)
        {
        to_point[0] = x1 + link_grth;
        }
 else
        {
        to_point[0] = x1 - link_grth;
        }
 return(to_point);
}
/*****************************************************************************

        SCENE 3: FUNCTION WHICH DEALS WITH LINK GROWTH IN THE SITUATION
                WHERE THE TARGET NODE AND THE LINK HEAD ARE NOT ALIGNED PARALLEL
                TO EITHER THE X OR Y AXIS.

*****************************************************************************/

int *scene3(int x1,int x2,int y1,int y2,int link_grth)
{
        double x3=0;                    double y3=0;

        int *to_point = NULL;

        double temp1=0.0;               double temp2=0.0;
        double dist_2_go=0.0;  double cos_theta=0.0;
        double slope=0.0;
 to_point = (int *)calloc(10,sizeof(int));

 if(to_point == NULL)
        {
        printf("Not enough memory to alloc. scene1\n\n ANY KEY TO TERMINATE !");
        getch();
        stall();
                /* TERMINATE THE PROGRAM GRACEFULLY*/
        }
 temp1 = pow((x2-x1),2);
 temp2 = pow((y2-y1),2);
 temp1 += temp2;
 dist_2_go = sqrt(temp1);
                /* THE FUNCTION get_distance IS NOT USED BECAUSE IT RETURNS AN */
                /* INTEGER VALUE.*/

 cos_theta = (x2-x1)/dist_2_go;
 x3 = (double)((link_grth * cos_theta) + x1);

                /* CALCULATION OF x3 USING PYTHAGOROUS AND BASIC TRIGONOMETERY*/

 slope = (double)((double)(y2-y1)/(double)(x2-x1));

 y3 =(double)( (slope*(x3-x1))+y1 );

 if(y3-ceil(y3) <= 0.5)
        to_point[1] = (int)y3;
 else
        to_point[1] = (int)(y3+1);

                /* CALCULATION OF y3 USING THE EQUATION OF THE LINE*/
                /* (Y-Y1) = (SLOPE)*(X-X1) AND CONVERSION TO TYPE int*/

 if(x3-ceil(x3) <= 0.5)
        to_point[0] = (int)x3;
 else
        to_point[0] = (int)(x3+1);
                /* ADJUSTMENT TO INTEGER VALUE FOR X3*/
 return(to_point);
}

/***************************************************************************

        GET POINT: THIS FUNCTION EVALUATES THE DISTANCE AND DIRECTION IN WHICH
                A LINK WILL GROW. THE PARAMETERS OF THE GROWTH ARE DECODED FROM
                GENES USING THE FUNCTIONS IN GET_DATA.C , THIS FUNCTION USES THE
                FUNCTIONS scene_n(); TO ACCOUNT FOR THE DIFFERENT RELATIVE
                POSITIONS OF NODES IN THE XY PLANE. THE DIRECTIONAL ANGLE OF THE
                LINK GROWTH MUST BE TAKEN INTO ACCOUNT.

***************************************************************************/

int *get_point(int grth_dis,int x1,int y1,int x2,int y2)
{
        int *to_point=NULL;

if ((x1==x2) && (y1!=y2))
                /* TWO POSITIONS ARE ALIGNED PARALLEL TO THE Y AXIS*/

        {
        to_point = scene1(x1,y1,y2,grth_dis);
        }

if ((y1==y2) && (x1 != x2))
                /* CASE: TWO POSITIONS ARE ALIGNED PARALLEL TO THE X AXIS*/

        {
        to_point = scene2(x1,x2,y1,grth_dis);
        }

if ((x1!=x2) && (y1 != y2))
                /* THE POSITIONS ARE NOT ALIGNED WITH EITHER AXIS*/

        {
        to_point = scene3(x1,x2,y1,y2,grth_dis);
        }

return(to_point);

}

/***************************************************************************

        MAKE_CONNECTION: FUNCTION TO COMPLETE THE 'PAPERWORK' WHEN A CONNECTION
                IS MADE BETWEEN TWO NODES

***************************************************************************/

void make_connection(int base_node,int nr_nd_num,int *inbt)
{
        int x1=0;               int y1=0;               int x2=0;
        int y2=0;            int distance=0;     int wgt_nm=0;

#if SHOW_NODE_0

 if(base_node==0)
        {
        getch();
        fflush(stdin);
        }

#endif

 conns[num_conns][0] = base_node;
 conns[num_conns][1] = nr_nd_num;
                /* UPDATE CONNECTION ARRAY*/

 links[base_node][0] = nodes[base_node][0];
 links[base_node][1] = nodes[base_node][1];
                /* RESET LINK ARRAY*/
 nodes[base_node][2]++;
           /* INCREMENT CONNECTION COUNT FOR BASE NODE*/

 num_conns++;
           /* INCREMENT THE NUMBER OF CONNECTIONS.*/

 x1=nodes[base_node][0];
 y1=nodes[base_node][1];

 x2=nodes[nr_nd_num][0];
 y2=nodes[nr_nd_num][1];

 distance = get_distance(x1,y1,x2,y2);
               /* CALCULATE THE DISTANCE BETWEEN THE NODES AS THIS FIGURE WILL   */
               /* WILL USED TO CALCULATE THE NEURAL NETWORK WEIGHT BETWEEN THE   */
               /* TWO NODES*/

 if(distance >=255)
       distance = 255;
               /* NORMALISE THE DISTANCE BETWEEN THE CONNECTED NODES*/

 wgt_nm = nodes[base_node][2];
               /* CHECK WHICH CONNECTION NUMBER IT IS AND CHECK AGAINST THE      */
               /* DECODED CONNECTIONS GENETIC PARAMETER TO SEE IF IT SHOULD BE   */
               /* A POSITIVE OR AN INHIBITORY LINK.*/

 if (inbt[wgt_nm-1]==0)
       distance *= -1;
 wghts[base_node][wgt_nm-1] = distance;
               /* UPDATE THE NETWORK CONNECTION WEIGHTS ARRAY*/

}

/***************************************************************************

       GROW_LINKS: CONTROLLING FUNCTION TO SIMULATE LINK GROWTH CALLED FROM
               THE DEVEL.C PROGRAM.

***************************************************************************/

void grow_links(int lnk_grt,int *inbt)
{
       int *nr_nodes=NULL;           int *to_point=NULL;

       int iter=0;                   int x=0;                int y=1;
       int nr_nd_num=0;              int x1=0;                     int y1=0;
       int x2=0;                         int y2=0;                         int x3=0;
       int y3=0;                 int nr_nd_dis=0;    int connect=0;
 for (iter=0;iter<num_nodes;iter++)
               /* EACH NODE IS POTENTIALLY GROWING AT THE SAME TIME*/

       {
       if ((iter == MOTOR_A) || (iter == MOTOR_B) );
               /* THE MOTOR NODES DO NOT GROW ANY LINKS -> THEY ARE PURELY */
               /* RECEPTIVE NODE.*/

       else
       {
       if(nodes[iter][2]<num_outputs)
                       /* CHECK THAT MAXIMUM ALLOWED NUMBER OF CONNECTIONS HAS NOT */
                       /* BEEN EXCEEDED BY NODE*/

               {
               x1 = links[iter][x];
               y1 = links[iter][y];
                               /* HEAD OF THE NODES GROWING LINK*/

               if ( (nr_nodes  = find_near(iter,0) ) != NULL )
                               /* FIND THE NEAREST NODE*/

                       {
```

```
                nr_nd_dis  = nr_nodes[0];
                nr_nd_num  = nr_nodes[1];
                free(nr_nodes);
                                /* MEMORY MANAGEMENT*/

                if (nr_nd_dis > lnk_grt)
                        /* IF THE DISTANCE FROM THE HEAD OF THE LINK TO THE     */
                        /* NODE IS GREATER THAN THE ALLOWED LINK GROWTH         */
                        /* DISTANCE THEN ENTER THIS SECTION*/

                        {
                        x2 = nodes[nr_nd_num][x];
                        y2 = nodes[nr_nd_num][y];
                        to_point  = get_point(lnk_grt,x1,y1,x2,y2);
                                /* CALL TO FUNCTION TO FIND OUT THE POSITION OF   /
                                /* THE HEAD OF THE LINK AFTER GROWTH.*/

                        x3 = to_point[x];
                        y3 = to_point[y];
                                /* POINT OF HEAD AFTER GROWTH */

                        free(to_point);
                                /* MEMORY MANAGEMENT*/

                        links[iter][x] = x3;
                        links[iter][y] = y3;
                                /* UPDATE THE LINKS ARRAY*/
                                }
                else
                        /* IF WITHIN A SINGLE GROWTH CYCLE DISTANCE OF TARGET   */
                        /* NODE*/

                        {
                        x3 = nodes[nr_nd_num][x];
                        y3 = nodes[nr_nd_num][y];
                        make_connection(iter,nr_nd_num,inbt);
                        connect = 1;
                                /* CALL TO FUNCTION TO CONNECT THE NODES */

                        }
#if SHOW_GROWTH

        if ( *(inbt+(nodes[iter][2])-connect) == 1)
                        /* IF POSITIVE CONNECTION*/
                {
                if(iter<NUM_SENSORS)
                        {
                        setcolor(LIGHTGREEN);
                        graphics_check(graphresult());
                        }
                else
                        setcolor(GREEN);
                                graphics_check(graphresult());

                if(connect == 1)
                    {
                    setcolor(YELLOW);
                    }

                connect=0;
                        /* RESET CONNECTION VARIABLE*/
                }

        else
                {

                if(iter<NUM_SENSORS)
                        {
                        setcolor(LIGHTBLUE);
                        graphics_check(graphresult());
                        }
                        else
                                setcolor(BLUE);
                                        graphics_check(graphresult());

                        if(connect == 1)
                            {
                            setcolor(YELLOW);
                            }

                        connect = 0;
                                /* RESET CONNECTION VARIABLE*/

                        }

#if SHOW_NODE_0
                if(iter==0)
                        {
                        setcolor(random(15)+1);
                                graphics_check(graphresult());

                        line(x1,y1,x3,y3);
                                graphics_check(graphresult());

                        }
#else
                line(x1,y1,x3,y3);
                        graphics_check(graphresult());
#endif /* {{SHOW_NODE_0}} */
#endif          /* {{SHOW_GROWTH}} */

                        }
                }
                }
        }
}
```

```
/*
      GRO_NODE.C

        THIS PROGRAM CONTROLS THE MANNER IN WHICH NODES ARE GROWN.

        IT IS CALLED BY THE PROGRAM DEVEL.C*/
#include "devel.h"
                /* STANDARD INCLUDE FILE     */

#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>

#define NODE_EXCLUSION_RANGE 20
#define LEFT   50
#define RIGHT  306
#define TOP    50
#define BOTTOM 306
                /* DEFINITIONS LOCAL TO THE GRO_NODE PROGRAM*/
/***************************************************************************/
/**/
/*      FIX_POSITION : FUNCTION TO ADJUST THE INTERSECTION POINT OF THE LINE*/
/*              JOINING THE TWO NEAREST NODES IF THE NODE AND ITS CLOSEST   */
/*              NEIGHBOURS ALL LIE IN THE SAME STRAIGHT LINE*/
/*                                                                          */
/***************************************************************************/

int *fix_position(int close1,int close2,int midx,int midy)
{
        int       *mid_fixed=NULL;

        int       x1,y1=0;     int       x2,y2=0;     int    x3,y3=0;
        int    x4,y4=0;        int       x=0;         int    y=1;
        int    distance1=0;    int    distance2=0;

        double slope=0.0;              double new_slope=0.0;

 if ((mid_fixed = (int *)malloc(sizeof(int)*100)) == NULL)
        {
        printf("Not enough memory to allocate mid_fixed\n");
        stall();
                /* TERMINATE PROGRAM GRACEFULLY*/
        }

 x1=midx;
 y1=midy;

 if(nodes[close1][y]==nodes[close2][y])
                /* IF LINED UP PARALLEL TO THE X-AXIS                              */

        {
        if(midy <=178)
                midy += 10;
        else
                midy -= 10;
                        /* ADJUST SO THAT THE MID-POINT MOVES TOWARD THE CENTRE*/

        *(mid_fixed+0)=midx;
        *(mid_fixed+1)=midy;
        return(mid_fixed);
                /* EXIT FUNCTION WITH ADJUSTED NUMBERS*/

        }

 if(nodes[close1][x]==nodes[close2][x])
                /* IF LINED UP PARALLEL TO THE Y-AXIS*/

        {
        if(midx<=178)
                midx += 10;
        else
                midx -= 10;
```

```
                        /* ADJUST SO THAT THE MID-POINT MOVES TOWARD THE CENTRE*/

        *(mid_fixed+0)=midx;
        *(mid_fixed+1)=midy;
        return(mid_fixed);
                /* EXIT FUNCTION WITH ADJUSTED VALUES*/
        }
 else
        {
        x1=nodes[close1][x];
        x2=nodes[close2][x];
        y1=nodes[close1][y];
        y2=nodes[close2][y];

        slope=(double)( (double)(y2-y1)/(double)(x2-x1));
                /* STANDARD EQUATION FOR THE SLOPE OF A LINE*/

        new_slope = (-1)/slope;
                /* SLOPE OF THE LINE PERPENDICULAR TO THE LINE JOINING THE TWO */
                /* CLOSEST NODES.*/

        x3 = midx + 100;
        y3 = (new_slope*(x3-midx))/(-midy);
                /* FROM THE EQUATION OF THE LINE*/
                /* ADJUST X AND CALCULATE THE CORRESPONDING VALUE BASED ON THE   */
                /* EQUATION OF THE LINE*/

        distance1=get_distance(x3,y3,178,178);
                /* CALCULATE THE DISTANCE FROM THE CENTRE OF THE GRID.*/

        x4 = x1 -100;
        y4 = (new_slope*(x4-x1))/(-y1);
                /* AS ABOVE BUT THE ADJUSTMENT TO X IS IN THE OPPOSITE DIRECTION */

        distance2=get_distance(x4,y4,178,178);
                /* CALCULATE THE DISTANCE FROM THE CENTRE OF THE GRID.           */

        if (distance1<=distance2)
                {
                *(mid_fixed+0)=x3;
                *(mid_fixed+1)=y3;
                }
        else
                {
                *(mid_fixed+0)=x4;
                *(mid_fixed+1)=y4;
                }
                        /* CHOOSE WHICHEVER SET OF POINTS ARE CLOSEST TO THE CENTRE */

        return(mid_fixed);
                /* RETURN THE ADJUSTED VALUES*/
        }
 }
/*****************************************************************************

        NEW_NODE_VALIDATE: FUNCTION TO CHECK IF NODE EXISTS IN A POSITION
                (x1,y1) ALREADY.

*****************************************************************************/

int new_node_validate(int total_node_number,int x1,int y1)
{
        int x2=0;               int y2=0;                       int iter = 0;
        int distance=0;

 for (iter=0;iter<total_node_number;iter++)
                /* CHECK ALL NODES*/

        {
        if ( (nodes[iter][0]==x1) & (nodes[iter][1]==y1) )
                /* IF IN EXACTLY THE SAME POSITION RETURN 0*/
                {
                return(0);
                }
        x2 = nodes[iter][0];
        y2 = nodes[iter][1];
        distance = get_distance(x1,y1,x2,y2);

        if (distance<=NODE_EXCLUSION_RANGE)
                {
                return(0);
                }
                        /* IF WITHIN EXCLUSIVE AREA THEN RETURN 0*/

        }
 return(1);
                /* ELSE RETURN 1*/
}

/*****************************************************************************

        CHECK_LOCATION_VALID: FUNCTION WHICH ENSURES THAT ANY NEW NODES CREATED
                ARE PLACED ON THE GROWTH GRID.
*****************************************************************************/

int *check_location_valid(int x2,int y2)
{
        int *division_pt=NULL;
 if ((division_pt = (int *)malloc(sizeof(int)*50)) == NULL)
        {
        printf("Not enough memory to allocate division_pt\n");
        stall();
                /* TERMINATE PROGRAM GRACEFULLY*/
        }
 if ( (x2>=LEFT) && (x2<=RIGHT) )
        *(division_pt)= x2;
 else
        {
        if (x2<LEFT)
                *(division_pt) = RIGHT - (LEFT-x2);

        while (x2>RIGHT)
                {
                *(division_pt) = LEFT + (x2-RIGHT);
                x2 -= RIGHT;
                }
        }

 if ( (y2>=TOP) && (y2<=BOTTOM) )
        *(division_pt+1) = y2;
 else
        {
        if (y2<TOP)
                *(division_pt+1) = BOTTOM-(TOP-y2);

        while (y2>306)
                {
                *(division_pt+1) = TOP + (y2-BOTTOM);
                y2 -=  BOTTOM;
                }
        }
 return(division_pt);
}

/*****************************************************************************

        GROW_NODES: FUNCTION TO SIMULATE THE GROWTH OF NODES.
                CALLED BY DEVEL.C

*****************************************************************************/

void grow_nodes(int div_ds)
```

```
{

        int *nr_nodes=NULL;  int *div_pt=NULL;      int *mid_fixed=NULL;

        int iter=0;                 int close1=0;           int close2=0;
        int midx=0;                 int midy=0;             int new_nodes=0;
        int x1=0,x2=0;          int y1=0,y2=0;      int start_node=0;
        int space_free = 0;

       char msg[10];

      static int fresh_nodes=7;

start_node = num_nodes-fresh_nodes-1;
              /* FIRST NODE FOR DIVISION IN CYCLE.*/
fresh_nodes=0;
              /* NEWLY CREATED NODES COUNTER*/
for (iter=start_node ;iter<num_nodes;iter++)
              /* LOOP NUMBER OF TIMES EQUAL TO NUMBER OF NEW NODES CREATED IN */
              /* THE LAST NODE DIVISION CYCLE*/

      {
      x1 = nodes[iter][0];
      y1 = nodes[iter][1];

      nr_nodes = find_near(iter,1);
      close1 = *(nr_nodes+1);
      close2 = *(nr_nodes+2);
      free(nr_nodes);
              /* FIND TWO NEAREST NODES TO THE DIVIDING NODE*/
              /* THIS IS BECAUSE THE LOCATION OF A NEW NODE LIES ON A PATH*/
              /* WHICH BISECTS THE CENTRE OF THE STRAIGHT LINE JOINING THE TWO */
              /* NODES NEAREST TO THE DIVIDING NODE.*/

      midx = (nodes[close1][0]+nodes[close2][0])/2;
      midy = (nodes[close1][1]+nodes[close2][1])/2;

      if((midx==x1)&&(midy==y1))
              {
              mid_fixed=fix_position(close1,close2,midx,midy);
              midx=*(mid_fixed+0);
              midy=*(mid_fixed+1);
              free(mid_fixed);
              }
                      /* CALL TO THIS SECTION NECCESSARY WHEN THE LINE JOINING    */
                      /* THE TWO NEAREST NODES TO THE DIVIDING NODE PASSES */
                      /* THROUGH THE NODE WHICH IS DIVIDING (IE ALL 3 INVOLVED    */
                      /* NODES LIE ALONG THE SAME STRAIGHT LINE.*/

      div_pt = get_point(div_ds,x1,y1,midx,midy);
      x2 = div_pt[0];
      y2 = div_pt[1];
      free(div_pt);
              /* GET POSITION OF NEWLY CREATED NODE ON THE GRID*/
              /* AND CLEAR UP MEMORY ALLOCATION*/

      div_pt = check_location_valid(x2,y2);
      x2 = div_pt[0];
      y2 = div_pt[1];
      free(div_pt);
              /* CHECK THAT THE RETURNED POSITION IS A VALID ONE AND IF NOT */
              /* THEN ADJUST IT TILL IT IS.*/
              /* ALSO CLEAR UP MEMORY AGAIN.*/

      space_free = new_node_validate(num_nodes+new_nodes,x2,y2);
              /* CALL TO FUNCTION TO CHECK IF THE CHOSEN SPOT ON THE GROWTH    */
              /* GRID IS ALREAD OCCUPIED OR WITHIN A RANGE OF 20 RADIAL UNITS  */
              /* OF THE NEAREST EXISTANT NODE*/

      if ( space_free )
              {
              nodes[num_nodes+new_nodes][0] = x2;
```

```
                nodes[num_nodes+new_nodes][1] = y2;
                links[num_nodes+new_nodes][0] = x2;
                links[num_nodes+new_nodes][1] = y2;
                        /* IF SPACE IS FREE THEN STORE POSITION OF NEW NODE ON THE  */
                        /* GRID AND ALSO INITIALISE THE LINK HEAD POSITION OF THE   */
                        /* NODE.*/

#if SHOW_GROWTH
                setcolor(WHITE);
                        graphics_check(graphresult());
                sprintf(msg,"%d",(num_nodes+new_nodes));
                        graphics_check(graphresult());

                outtextxy(x2+5,y2,msg);
                        graphics_check(graphresult());

                putpixel(x2,y2,LIGHTBLUE);
                        graphics_check(graphresult());

                rectangle(x2-1,y2-1,x2+1,y2+1);
                        graphics_check(graphresult());

                rectangle(x2-2,y2-2,x2+2,y2+2);
                        /* PRINT NEW NODES ON-SCREEN*/
                        graphics_check(graphresult());

                setcolor(RED);
                        graphics_check(graphresult());
#endif  /* {SHOW_GROWTH} */

                new_nodes++;
                        /* UPDATE NEW NODE COUNT FIGURE*/

                }
        }
 num_nodes += new_nodes;
                /* UPDATE TOTAL NODE COUNT FIGURE*/

 fresh_nodes=new_nodes;
                /* STORE NUMBER OF NEW NODES CREATED FOR NEXT TIME.*/

}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "devel.h"

extern int nodes[MAX_NODES][3];              /* Global Variables Declaration */
extern int links[MAX_NODES][2];              /* and Initialisation           */
extern int conns[MAX_NODES*MAX_NO_OP][2];
extern int wghts[MAX_NODES][10];

void outter(void)
{
        FILE *stream;
        int i;
        int base_node;
        int targ_node;

stream = fopen("dum.fil","w");

fprintf(stream,"\t%s\n","NODE    TO    WGT");

for(i=0;i<MAX_NODES;i++)
   {
       base_node = links[i][0];
       targ_node = links[i][1];
       fprintf(stream,"\n\t %d      %d",base_node,targ_node);
   }

fclose(stream);
}
```

Appendix F2 Simple Genetic Algorithm Code

```
/***************************************************************************

 PROGRAM SGA.C
        THIS PROGRAM OPERATES ON A GENETIC ALGORITHM BASE.
        IT USES A NUMBER OF PROGRAMS TO ENABLE IT TO DO THIS:

        CROSSER.C          RANDOMS.C       OUTPOP.C

        EACH OF THESE PROGRAMS IS EXPLAINED IN THEIR OWN FILES.

        THE PROGRAM PROMPTS THE USER FOR TEN FITNESS VALUES AND THEN
        USES THESE VALUES TO CREATE A NEXT GENERATION OF INDIVIDUALS
        BASED ON PROBABILISTIC TRANSITION RULES.

***************************************************************************/

#define CROSS 1
               /* USED BY THE PREPROCESSOR (SEE SGA.H FOR EXPLANATION)        */

#include "sga.h"
               /* STANDARD HEADER FILE       */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <graphics.h>

#if DEBUG
        extern int mutate_record[420];
               /* USED FOR ANALYSIS PURPOSES */
#endif

struct individual
       {
         int chromosome[GENE_LENGTH];
         int fitness;
       };
               /* EACH INDIVIDUAL IS CONSTRUCTED OF TWO PARTS:                */
               /* A: THE CHROMOSOME                                           */
               /* B: THE FITNESS FIGURE ASSIGNED TO IT BY THE USER            */

struct individual new_population[MAXPOP_SIZE];
struct individual old_population[MAXPOP_SIZE];
               /* ARRAYS CONTAINING THE ENCODED GENOTYPES OF THE OLD AND NEW  */
               /* GENERATIONS
                */

double total_genetic_fitness;
int popsize=0;

/****************************************************************************

       STALL: FUNCTION TO TERMINATE PROGRAM GRACEFULLY IN THE EVENT OF ERRORS
              OCCURING.

****************************************************************************/

void graphics_check(int errorcode)
{
if (errorcode != grOk)
       {
       closegraph();
       printf("GRAPHICS ERROR : %s \n",grapherrormsg(errorcode));
       printf("\n\n\Any Key to Exit !!");
       exit(1);
       }
```

```
}

/****************************************************************************

        SET_GRAPHICS: FUNCTION TO INITIALISE GRAPHICS.

****************************************************************************/

void set_graphics(void)
{
        int gdriver = DETECT, gmode, errorcode;
                /* REQUEST AUTO DETECTION */

#if DDRIVE

 initgraph(&gdriver, &gmode, "D:\\AUTO");
                /* INITIALISE GRAPHICS MODE */
#else
 initgraph(&gdriver, &gmode, "C:\\AUTO");
                /* INITIALISE GRAPHICS MODE */

#endif

 errorcode = graphresult();
                /* READ RESULT OF INITIALISATION */

 if (errorcode != grOk)
                /* AN ERROR OCCURED */

        {
        printf("Graphics error: %s\n", grapherrormsg(errorcode));
        printf("Press any key to halt:");
        exit(1);
                /* RETURN WITH ERROR CODE */
        }

 setfillstyle(0,0);
                /* SET THE FILL PATTERN AND COLOUR */
        graphics_check(graphresult());

 cleardevice();
                /* CLEAR SCREEN AND RETURN 'CURSOR' POSITION TO (0,0)*/
        graphics_check(graphresult());

 setcolor(WHITE);
                /* SET FOREGROUND COLOUR */

}

/***************************************************************************

        FUNCTION WHICH PROMPTS USER FOR 10 FITNESS FIGURES

***************************************************************************/

int getfit(void)
{
        int iter;
        int fit;

        FILE *ofp;

 if ((ofp = fopen("C:\\AUTO\\data\\GENETIC.RES", "r+")) == NULL)
        {
        fprintf(stderr, "Cannot open input file genetic.res.\n");
        exit(1);
        }
 fseek(ofp,SEEK_SET,0);
 for(iter=0;iter<MAXPOP_SIZE;iter++)
        {
        fscanf(ofp,"%d",&fit);
        if ( (fit<0) || (fit>100))
```

```
                fit=5;
        old_population[iter].fitness = (int)fit;
        total_genetic_fitness += fit;
        }

fclose(ofp);
return(1);
}

/****************************************************************************

        FUNCTION TO DISPLAY THE OLD POPULATION AND CONTROL THE
        OBTAINING OF FITNESS VALUES.

****************************************************************************/

void getpop(void)
{
        FILE *f_point;
        int iter;
        int itrb;
        int i=0;
        int number=0;
        char msg[10];

if ((f_point = fopen("c:\\auto\\data\\POPDATA.FIL", "rt+")) == NULL)
        {
        fprintf(stderr, "Cannot open data file popdata.fil.\n");
        exit(1);
        }

fseek(f_point,SEEK_SET, 0);
for (number = 1 ; number<=MAXPOP_SIZE; number++)
        {
        for (iter = 0; iter<GENE_LENGTH; iter++)
                {
                if (fscanf(f_point, "%1i", &i))
                        old_population[number-1].chromosome[iter] = i;
                else
                        {
                        fprintf(stderr, "Error reading from POPDATA.FIL !!\n");
                        exit(1);
                }
        }
}

while ( !getfit() );

printf(" \t\t\t PARENT CHROMOSOMES \t\t\t  FITNESS\n");
for (itrb=0;itrb<MAXPOP_SIZE;itrb++)
        {
        for (iter=0;iter<GENE_LENGTH;iter++)
                {
                sprintf(msg,"%d",old_population[itrb].chromosome[iter]);
                outtextxy((iter*9)+20,(itrb*20)+60,msg);

                }

        sprintf(msg,"%d",old_population[itrb].fitness);
        outtextxy((GENE_LENGTH*9)+40,(itrb*20)+60,msg);

        }

outtextxy(30,500," \t\t\tPRESS SPACE KEY TO CONTINUE");
                /* Section to display the old poulation */

getch();

}

/****************************************************************************
```

```
        MAIN FUNCTION

*************************************************************************/

int main(void)
{
#if DEBUG
        clrscr();
        printf("debug");
        getch();
#endif

clrscr();
set_graphics();
getseed();
getpop();
closegraph();
crspop();
outpop();
out_generation_info();
saveseed();

return(1);
}
```

```
#define MAXPOP_SIZE     20
#define GENE_LENGTH     54
               /* CHROMOSOME IS 51 BITS LONG */

#define PROB_CROSSOVER  0.9
               /* PROBABILITY OF TWO CHOSEN MATES AFFECTED BY THE CROSSOVER
                  OPERATOR */

#define PROB_MUTATATION 0.01
               /* PROBABILITY OF ON AVERAGE 8 BITS MUTATING PER POPULATION
                  PER GENERATION */

#define DEBUG 0

#ifdef CROSS
       typedef  char   flag;          /* >=  1 bit, used as boolean  */
#endif
               /* THE TYPEDEF FLAG IS USED IN THE FUNCTION FLIP() AND IT
                  IS NECCESSARY TO INCLUDE THIS PREPROCCESSOR DIRECTIVE
                  TO PREVENT MULTIPLE DECLARATIONS OF THE FLAG TYPEDEF WHICH
                  RESULTS IN ERROR USING TURBO C++ COMPILER */

flag   flip(double);
double gen_float(void);
void   crspop(void);
void   outpop(void);
int    get_cross_point(int);
void   saveseed(void);
void   getseed(void);
void   out_generation_info(void);
```

```
/******************************************************************************

 CROSS.C

        PROGRAM WHICH IMPLEMENTS THE CROSSOVER OPERATOR UPON TWO
        DISTINCT INDIVIDUALS IN A POPULATION AND WHICH IS ITERATED
        UNTIL THE NEXT POPULATION GENERATION HAS BEEN PRODUCED. iT MAKES USE
        OF THE RANDOM PROCES FUNCTIONS DEFINED IN RANDOMS.C

        IT IS CALLED FROM THE PROGRAM SGA.C

******************************************************************************/

#define CROSS 1
                /* USED BY THE PREPROCCESSOR */

#include "sga.h"
                /* STANDARD HEADER FILE */

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <conio.h>
#include <graphics.h>

extern struct individual
        {
          int chromosome[GENE_LENGTH];
          int fitness;
        };
extern struct individual new_population[MAXPOP_SIZE];
extern struct individual old_population[MAXPOP_SIZE];
extern double total_genetic_fitness;
extern int    popsize;
                /* THESE DATA VARIABLES ARE DEFINED ORIGINALLY IN SGA.C */

double random_floats[MAXPOP_SIZE];

#if DEBUG
        int mutate_record[420]={0};
                /* used for analysis purposes */
#endif

/******************************************************************************

        Function to implement the mutation operator

******************************************************************************/

int mutation(int single_gene)
{
        int new_gene;
        int mutate;

        #if DEBUG
                static int mutate_number=0;
                        /* used for analysis purposes */
        #endif

mutate = flip(PROB_MUTATATION);
                /* FLIP SIMULATES A WEIGHTED COIN TOSS AND IS USED */
                /* HERE TO DETERMINE WHETHER A SINGLE BIT SHOULD BE */
                /* MUTATED OR NOT.  */

#if DEBUG
        mutate_record[mutate_number]=mutate;
#endif

if (mutate)
        {
        textcolor(RED);
        if (single_gene == 1)
```

```
                new_gene = 0;
        else
                new_gene = 1;
        }
                /* IF THE FLIP FUNCTION RETURNS A '1' THEN THE BIT IN         */
                /* QUESTION IN THE CHROMOSOME IS INVERTED                     */
else
        new_gene = single_gene;

#if DEBUG
        mutate_number++;
#endif

return(new_gene);
                /* RETURN THE VALUE OF THE NEW BIT                                */
}

/****************************************************************************

        Function that implements the crossover operator upon two
        distinct individuals.

****************************************************************************/

void crossover(int mate1,int mate2,int crosspoint)
{
        int cross;     int xsite;     int iter;

        int parent1[GENE_LENGTH];     int parent2[GENE_LENGTH];

for(iter=0;iter<GENE_LENGTH;iter++)
        {
        parent1[iter] = old_population[mate1].chromosome[iter];
        parent2[iter] = old_population[mate2].chromosome[iter];
        }
                /* GET THE TWO CHROMOSOMES CHOSEN FOR REPRODUCTION OR CROSSING */
                /*  FROM THE OLD POPULATION RECORD                             */

cross = flip(PROB_CROSSOVER);
                /* FLIP SIMULATES A WEIGHTED 'COIN TOSS' AND IS USED HERE TO   */
                /* DETERMINE WHETHER THE TWO CHOSEN CHROMOSOMES SHOULD BE      */
                /* CROSSDE TOGETHER OR NOT */

if (cross)
        {
        xsite =  crosspoint;
        }
else
        xsite = GENE_LENGTH;
                /* IF THE CHROMOSOMES ARE NOT TO BE CROSSED THEN THE CROSSING  */
                /* SITE CHOSEN IS SIMPLY THE END OF THE TWO CHROMOSOMES        */

/*
gotoxy(xsite+7,(popsize*2) + 4);
printf("x");  */
                /* PRINT POSITION OF CROSSING SITE ON SCREEN                          */

for (iter=0;iter<xsite;iter++)
    {
    textcolor(GREEN);
    new_population[popsize].chromosome[iter] = mutation(parent1[iter]);
    gotoxy(iter+7,(popsize)+3);
    cprintf("%d",new_population[popsize].chromosome[iter]);
    textcolor(WHITE);
    new_population[popsize+1].chromosome[iter] = mutation(parent2[iter]);
    gotoxy(iter+7,(popsize)+4);
    cprintf("%d",new_population[popsize+1].chromosome[iter]);

    }
                /* THIS LOOP GENERATES TWO NEW MEMBERS OF THE POPULATION AND   */
                /* AS EACH BIT OF THE NEW CHROMOSOMES IS GENERATED BY THE      */
                /* CROSSING IT IS DETERMINED WHETHER OR NOT IT SHOULD BE       */
                /* MUTATED                                                          */

if(xsite!= GENE_LENGTH)
        {
        for(iter=xsite;iter<GENE_LENGTH;iter++)
                {
                textcolor(WHITE);
                new_population[popsize].chromosome[iter]=mutation(parent2[iter]);
                gotoxy(iter+7,(popsize)+3);
                cprintf("%d",new_population[popsize].chromosome[iter]);
                textcolor(GREEN);
                new_population[popsize+1].chromosome[iter]=mutation(parent1[iter]);
                gotoxy(iter+7,(popsize)+4);
                cprintf("%d",new_population[popsize+1].chromosome[iter]);
                gotoxy(5,popsize+3);

                }
        }
                /* IF THE CHROMOSOMES HAVE BEEN CHOSEN FOR CROSSING THEN THIS  */
                /* WILL IMPLEMENT THE ACTUAL EXCHANGE OF BITS. IF NOT THEN     */
                /* THE PROGRAM WILL NOT ENTER THIS LOOP                                */

}

/****************************************************************************

        FUNCTION WHICH IMPLEMENTS THE REPRODUCTION OPERATOR.

****************************************************************************/

int *select(double pair_number)
{
        int    j=0;
        int    i=0;
        int    roulette=0;
        double partsum=0;
        int    *mates=NULL;

if ((mates = malloc( 3*sizeof(int))) == NULL)
        {
        printf("Not enough memory to allocate buffer {fuction select()} \n");
        exit(1);
           /* TERMINATE PROGRAM IF OUT OF MEMORY */
}

for(i=0;i<2;i++)
        {
        roulette=(int)(random_floats[i+(pair_number*2)]* total_genetic_fitness);
                /* VALUE CHOSEN WHICH DETERMINES WHICH CHROMOSOME OF THE OLD   */
                /* POPULATION TO CROSS (OR NOT) WITH ANOTHER OLD CHROMOSOME.   */
                /* SEE GOLDBERG FRO DETAILS                                    */

        for(j=0;j<MAXPOP_SIZE;j++)
                {
                if((partsum += old_population[j].fitness)> roulette)
                        break;
                if( j == MAXPOP_SIZE-1)
                        break;
                }

        *(mates+i)=j;
        partsum=0;
        }

return(mates);
}

/****************************************************************************

        CONTROLLING FUNCTION TO CREATE AND DISPLAY A NEW GENERATION OF
        INDIVIDUALS.
```

```
*****************************************************************************/

void crspop()
{
        int mate1;      int mate2;      int iter;
        int fit1,fit2;
        int crosspoints[MAXPOP_SIZE/2];
        int *mates;

        FILE *ofp=NULL;

if ((ofp = fopen("c:\\auto\\data\\GN_HIST.FIL", "w+")) == NULL)
        {
        fprintf(stderr, "Cannot open GN_HIST.FIL for writing.\n");
        exit(1);
        }
                /*      GEN_HIST.FIL CONTAINS THE INFORMATION ON THE POPULATION */
                /*      HISTORY
             */

fseek(ofp,0L,SEEK_END);

clrscr();
printf("\t\t\tNEW GENERATION \t\t\t\t PARENTS(Fitness)\n");

for(iter=0;iter<MAXPOP_SIZE;iter++)
        {
        random_floats[iter]=gen_float();
        }
                /* GENERATION OF NUMBERS BETWEEN 1 & 0 TO BE USED IN THE       */
                /* select() FUNCTION FOR THE EVALUATION OF THE ROULETTE         */
                /* VARIABLE. IT IS A GLOBAL ARRAY. */

for(iter=0; iter<(MAXPOP_SIZE/2); iter++)
        crosspoints[iter]= get_cross_point(GENE_LENGTH);
                /* ALL CROSSING POINTS ARE CHOSEN AT THE SAME TIME AND THEN    */
                /* PASSED TO THE RELEVANT FUNCTION RATHER THAN EACH POINT      */
                /* BEING CHOSEN AS EACH PAIR OF CHROMOSOMES IS CROSSED         */

for (iter=0;iter<MAXPOP_SIZE;iter++)
        {
        if (popsize == MAXPOP_SIZE)
                break;
        else
                {
                mates = select(iter);
                mate1 = mates[0];
                mate2 = mates[1];
                fit1  = old_population[mate1].fitness;
                fit2  = old_population[mate2].fitness;

                        /* SELECT TWO CHROMOSOMES FROM THE OLD POPULATION FOR      */
                        /* CROSSING BASED ON THE FITNESS FIGURES                   */

                crossover(mate1,mate2,crosspoints[iter]);
                        /* CROSS THEM.
                   */

                fprintf(ofp,"\n%5d,%3d %9d (%2d) , %10d (%2d) %10d",
                        (iter*2)+1,(iter*2)+2,mate1+1,fit1,
                        mate2+1,fit2,crosspoints[iter]);

/*
                sprintf(msg,"%d",old_population[itrb].chromosome[iter]);
                outtextxy((iter*9)+20,(itrb*20)+60,msg);  */

                gotoxy(2,(iter*2)+3);
                printf("%d",(iter*2)+1);
                gotoxy(GENE_LENGTH+9,(iter*2)+4);
                printf("%d (%d), %d (%d)",mate1+1,fit1,mate2+1,fit2);
                gotoxy(2,(iter*2)+4);
                printf("%d",(iter*2)+2);
```

```
                        /* PRINT THE TWO NEW POPULATION MEMBERS ON THE VDU          */

                popsize += 2;
                        /* INCREASE THE RECORD OF NEW POPULATION MEMBERS            */

                }
                /* LOOP WHICH CONTROLS TEH GENERATION AND DISPLAY OF THE TWO    */
                /* NEW POPUALTION MEMBERS                                       */
        }
printf("\n\n\n\t\t\tPRESS ANY KEY TO CONTINUE ");
while (!kbhit());
free(mates);
fclose(ofp);
                /* MEMORY MANAGEMENT
           */
}
```

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>
#include <time.h>
#include <dos.h>
#include <stdlib.h>
#include <string.h>

#define TEST_TIME 240
               /* IN SECONDS*/

#define MAX_POP_SIZE 20

int fitness_scores[MAX_POP_SIZE];

int fitness=5;

int individual_number;

FILE *ofp=NULL;

/***************************************************************************

        OUTPUT_FITNESS_VALUE

***************************************************************************/

void output_fitness_value(void)

{
int iter;

 fseek(ofp,SEEK_SET,0);
 if (fitness < 5)
        fitness = 5;

 fitness_scores[individual_number-1]=fitness;

 for (iter=0;iter<MAX_POP_SIZE;iter++)
        {
        fprintf(ofp,"%d\n",fitness_scores[iter]);
        }

 fclose(ofp);

 exit(1);

}

/***************************************************************************

        Set_Up_Screen();

***************************************************************************/

int Set_Up_Screen(int number)
{
         int individual_number=MAX_POP_SIZE+1;
        char string[10];

 clrscr();

 textcolor(LIGHTBLUE);
 gotoxy(34,5);
 cprintf("%s"," FITNESS : ");

 textcolor(WHITE);
 cprintf("%d \n\n",fitness);

 textcolor(LIGHTBLUE);

 individual_number=number;
 gotoxy(25,3);
 cprintf("%s%d","CHROMOSOME NUMBER : ",
              individual_number);

 while( (individual_number<1) || (individual_number>MAX_POP_SIZE))
        {
        gotoxy(19,3);
        cprintf("%s","ENTER CHROMOSOME NUMBER :                        ");
        gotoxy(45,3);
        individual_number = atoi(gets(string));
        }
 return(individual_number);

}

/****************************************************************************

        Open_file();

****************************************************************************/

void open_file(void)
{
int iter;
int fit;

 if ((ofp = fopen("C:\\AUTO\\data\\GENETIC.RES", "r+")) == NULL)
        {
        fprintf(stderr, "Cannot open input file genetic.res .\n");
        exit(1);
        }
 fseek(ofp,SEEK_SET,0);
 for(iter=0;iter<MAX_POP_SIZE;iter++)
        {
        fscanf(ofp,"%d",&fit);
        fitness_scores[iter]=fit;
        }

}

/****************************************************************************

        END_TESTING();

****************************************************************************/

void end_testing(char *input_string)
{
char *test_string  = "time";
char *test_string2 = "terminate";
        clrscr();

        gotoxy(23,9);

        if (strcmp(test_string,input_string)==0)
                {
                cprintf("%s","           Time is Up");
                if (fitness > 100)
                        fitness = 100;
                }
        else
                {
                if (strcmp(test_string2,input_string)==0)
                        {
                        cprintf("%s","   Testing Terminated");
                        if (fitness > 100)
                                fitness = 100;
                        }
                else
                        cprintf("%s","Fitness Has Reached Base Level (5)");
                }
        gotoxy(25,11);
        cprintf("%s","Further Testing is Unneccessary");
        gotoxy(27,13);
        cprintf("%s","Press Any Key to Exit ....");
        getch();

        output_fitness_value();

}

/***************************************************************************

        PRINTSTRING

***************************************************************************/

void printstring(char *str1,char *str2,char *str3,char *str4)
{

 textcolor(WHITE);
 cprintf("%s",str1);
 textcolor(LIGHTBLUE);
 cprintf("%s",str2);
 textcolor(WHITE);
 cprintf("%s",str3);
 textcolor(LIGHTBLUE);
 cprintf("%s",str4);
 textcolor(WHITE);
 fflush(stdout);

}

/***************************************************************************

        PRINT_FITNESS

***************************************************************************/

void print_fitness(void)
{
char *fit="fitness";

 textcolor(LIGHTBLUE);
 gotoxy(34,5);
 cprintf("%s"," FITNESS :       ");

 if (fitness < 5)
        {
        end_testing(fit);
        }

 textcolor(WHITE);
 gotoxy(45,5);
 cprintf("%d",fitness);

 textcolor(LIGHTBLUE);
 fflush(stdin);
 fflush(stdout);

}

/***************************************************************************

        Set_Up_Screen_2();

***************************************************************************/

void Set_Up_Screen_2(void)
{

 clrscr();
 print_fitness();
 gotoxy(25,3);
 cprintf("%s","CHROMOSOME NUMBER : ");
 textcolor(LIGHTGREEN);
 cprintf("%d",individual_number);
 textcolor(LIGHTBLUE);
}

/****************************************************************************

       RECORD_FITNESS

****************************************************************************/

void record_fitness(int score,int x,int y)
{
 char input;

 input = tolower(getch());

 while ( (input != 'y') && (input != 'n'))
       input = tolower(getch());

 textcolor(WHITE);
 cprintf("%c",input);

 if (input  == 'n');
 else
       fitness += score;

 print_fitness();

 gotoxy(x,y);
}

/****************************************************************************

       EVALUATE_SENSORS();

****************************************************************************/

void evaluate_sensors(void)
{

 cprintf("%s","\r\n\n\n       MOVEMENT WITHOUT STIMULATION Y/N ? : ");
 record_fitness(10,wherex(),wherey());

 printstring("\r\nFRONT ","SENSORS ","TOGETHER ","OPERATIONAL  Y/N ? : ");
 record_fitness(7,wherex(),wherey());

 printstring("\r\nFRONT ","SENSORS ","DIRECTION ","CORRECT     Y/N ? : ");
 record_fitness(7,wherex(),wherey());

 printstring("\r\nFRONT ","SENSOR NUMBER ","1 ","OPERATIONAL   Y/N ? : ");
 record_fitness(7,wherex(),wherey());

 printstring("\r\nFRONT ","SENSOR ","1 DIRECTION ","CORRECT    Y/N ? : ");
 record_fitness(7,wherex(),wherey());

 printstring("\r\nFRONT ","SENSOR NUMBER ","2 ","OPERATIONAL   Y/N ? : ");
 record_fitness(7,wherex(),wherey());

 printstring("\r\nFRONT ","SENSOR ","2 DIRECTION ","CORRECT    Y/N ? : ");
 record_fitness(7,wherex(),wherey());

 printstring("\r\n REAR ","SENSORS ","TOGETHER ","OPERATIONAL  Y/N ? : ");
 record_fitness(7,wherex(),wherey());

 printstring("\r\n REAR ","SENSORS ","DIRECTION ","CORRECT     Y/N ? : ");
 record_fitness(7,wherex(),wherey());

 printstring("\r\n REAR ","SENSOR NUMBER ","3 ","OPERATIONAL   Y/N ? : ");
 record_fitness(3,wherex(),wherey());
 printstring("\r\n REAR ","SENSOR ","3 DIRECTION ","CORRECT    Y/N ? : ");
 record_fitness(3,wherex(),wherey());

 printstring("\r\n REAR ","SENSOR NUMBER ","4 ","OPERATIONAL   Y/N ? : ");
 record_fitness(3,wherex(),wherey());

 printstring("\r\n REAR ","SENSOR ","4 DIRECTION ","CORRECT    Y/N ? : ");
 record_fitness(3,wherex(),wherey());

 printstring("\r\n LEFT ","SENSOR NUMBER ","5 ","OPERATIONAL   Y/N ? : ");
 record_fitness(5,wherex(),wherey());

 printstring("\r\nRIGHT ","SENSOR NUMBER ","6 ","OPERATIONAL   Y/N ? : ");
 record_fitness(5,wherex(),wherey());

}

/****************************************************************************

        ON_LINE_TESTING

****************************************************************************/

void on_line_test(clock_t start_clk_time)
{
char answer;
char *out_of_time="time";
char *terminate="terminate";

 while( (clock()-start_clk_time)/CLK_TCK < TEST_TIME )
        {
 gotoxy(27,1);
 cprintf("SECONDS ELAPSED : %f\n",(clock()-start_clk_time)/CLK_TCK);

        if(kbhit())
                {
                answer = tolower(getch());

                        if( (answer == 't') || (answer == 's'))
                                {
                                fitness -= 10;
                                print_fitness();
                                }
                        else
                                ;
                        if( answer == 'q' )
                                end_testing(terminate);
                        else
                                ;

                while(fflush(stdin) != 0)
                        ;
                }

        }
 end_testing(out_of_time);

}

/****************************************************************************

        MAIN FUNCTION

****************************************************************************/

void main(int argc, char *argv[])
{

clock_t start_clk_time;

 open_file();
```

```
    individual_number=Set_Up_Screen(atoi(argv[1]));

    start_clk_time= clock();

    evaluate_sensors();

    Set_Up_Screen_2();

    on_line_test(start_clk_time);

}
```

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include "minimum.h"

struct individual
           {
           char chromosome[GENE_LENGTH];
           int fitness;
           };
               /* EACH INDIVIDUAL IS CONSTRUCTED OF TWO PARTS:                 */
               /* A: THE CHROMOSOME                                            */
               /* B: THE FITNESS FIGURE ASSIGNED TO IT BY THE USER             */

struct individual pop[MAXPOP_SIZE];

void compare(void)
{

FILE *f_point;
int iter;
int itrb;
int already[20]={00000000000000000000};
int index=0;
int to___do[20];
char *buf1=NULL;
char *buf2=NULL;
int res=0;

if ((f_point = fopen("C:\\AUTO\\data\\minimum.fil","w+"))
        == NULL)
        {
        fprintf(stderr, "Cannot open input file.\n");
        exit(1);
        }

fprintf(f_point,"\n\n TO DO: ");

for(iter=1;iter<MAXPOP_SIZE+1;iter++)
        {
        for(itrb=1;itrb<MAXPOP_SIZE+1;itrb++)
                {
                if (iter == already[itrb-1])
                   {
                   res +=1;
                   }
                }
        if(res==0)
                {
                fprintf(f_point,"\n %3d,",iter);
                buf1=pop[iter-1].chromosome;

                for (itrb=1;itrb<MAXPOP_SIZE+1;itrb++)
                        {
                        if (itrb != iter)
                                {

                                buf2=pop[itrb-1].chromosome;
                                if((strcmp(buf1,buf2))== 0)
                                        {
                                        fprintf(f_point," %3d,",itrb);
                                        already[index]=itrb;
                                        index++;
                                        }
                                }
                        }
                }
        else
                res=0;
        }
}

/*****************************************************************************

        FUNCTION TO DISPLAY THE OLD POPULATION AND CONTROL THE
        OBTAINING OF FITNESS VALUES.

*****************************************************************************/

void getpop(void)
{
        FILE *f_point;
        int iter;
        int itrb;
        char c;
        int number=0;
        char msg[10];

if ((f_point = fopen("c:\\auto\\data\\POPDATA.FIL", "rt+")) == NULL)
        {
        fprintf(stderr, "Cannot open data file popdata.fil.\n");
        exit(1);
        }

fseek(f_point,SEEK_SET, 0);
for (number = 1 ; number<=MAXPOP_SIZE; number++)
        {
        if ( fscanf(f_point, "%s", pop[number-1].chromosome) == 0 )
           printf("\nUnable to read in popdata.fil data");

         }
}

void main(void)
{
getpop();
compare();
}
```

```
/***************************************************************************

 OUTPOP.C

        PROGRAM TO OUTPUT THE NEW GENERATION OF INDIVIDUALS.

        CALLED BY SGA.C

***************************************************************************/

#define CROSS 1
               /* USED BY PREPROCESSOR */

#include "sga.h"
               /* STANDARD HEADER FILE */

#include <stdio.h>
#include <stdlib.h>

extern struct individual
       {
         int chromosome[GENE_LENGTH];
         int fitness;
       };

extern struct individual new_population[MAXPOP_SIZE];
extern struct individual old_population[MAXPOP_SIZE];
               /* INITIAL DEFINITIONS CONTAINED IN SGA.C */

/***************************************************************************

        CONTROLLING FUNCTION CALLED FROM SGA.C.

***************************************************************************/

void outpop()
{
        FILE *f_point;
        int iter;
        int number=0;
        int gene[GENE_LENGTH] = {0};

if ((f_point = fopen("c:\\auto\\data\\POPDATA.FIL", "r+")) == NULL)
        {
        fprintf(stderr, "Cannot open POPDATA.FIL for writing.\n");
        exit(1);
        }
               /* POPDATA.FIL CONTAINS THE INFORMATION ON THE TEN CHROMOSOMES */

fseek(f_point,SEEK_SET, 0);

for (number = 1 ; number<=MAXPOP_SIZE ; number++)
        {
        for (iter=0;iter<GENE_LENGTH;iter++)
               {
               gene[iter]= new_population[number-1].chromosome[iter];
               fprintf(f_point,"%d",gene[iter]);
               }
        if (number!=MAXPOP_SIZE)
               fprintf(f_point,"\n");
        }
fclose(f_point);
}

void out_generation_info(void)
{
FILE *gen_data=NULL;

int iter;
int itrb;
int total=0;
int topfit=0;
int subconverg_a=0.0;
int subconverg_b=0.0;

double average;
double dummy;
double convergence=0.0;

gen_data=fopen("C:\\auto\\data\\GEN_DATA.FIL","rt+");

fseek(gen_data,0,SEEK_END);

fprintf(gen_data,"\n");

for(iter=0;iter<MAXPOP_SIZE;iter++)
   {
   if(old_population[iter].fitness > topfit)
               topfit = old_population[iter].fitness;

   fprintf(gen_data,"%2d'",old_population[iter].fitness);
   total += old_population[iter].fitness;

   }

for(itrb=0;itrb<GENE_LENGTH;itrb++)
   {
   subconverg_a = 0.0;
   subconverg_b = 0.0;

   for(iter=0;iter<MAXPOP_SIZE;iter++)

          {
          if (old_population[iter].chromosome[itrb] == 1)
                 subconverg_a += 1;
          else
                 subconverg_b += 1;
         }
  dummy = (double)(subconverg_a - subconverg_b)/MAXPOP_SIZE;
  if (dummy <0)
          dummy *= -1;

  convergence += dummy;

  }

convergence /= GENE_LENGTH;
average = (double)((double)total/(double)MAXPOP_SIZE);
fprintf(gen_data," %3.2f'",average);
fprintf(gen_data," %2d'",topfit);
fprintf(gen_data," %.4f' ",convergence);

fclose(gen_data);

}
```

```
#include <stdio.h>
#include <string.h>

#define I 555
#define R 5.5

int main(void)
{
   int i,j,k,l;
   char buf[7];
   char *prefix = buf;
   char tp[20];
   printf("prefix  6d      6o      8x        10.2e         "
          "10.2f\n");
   strcpy(prefix,"%");
   for (i = 0; i < 2; i++)
   {
      for (j = 0; j < 2; j++)
         for (k = 0; k < 2; k++)
            for (l = 0; l < 2; l++)
            {
               if (i==0)  strcat(prefix,"-");
               if (j==0)  strcat(prefix,"+");
               if (k==0)  strcat(prefix,"#");
               if (l==0)  strcat(prefix,"0");
               printf("%5s |",prefix);
               strcpy(tp,prefix);
               strcat(tp,"6d |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"6o |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"8x |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"10.2e |");
               printf(tp,R);
               strcpy(tp,prefix);
               strcat(tp,"10.2f |");
               printf(tp,R);
               printf("  \n");
               strcpy(prefix,"%");
            }
      }
   return 0;
}
```

Appendix F3 Neural Network Simulator Code

```
/*

      PROGRAM WHICH ACTS AS SIMULATOR FOR A NEURAL NETWORK DESIGNED

         USING THE REDUCED COMPLEXITY NEURAL NODE MODEL AND GENETIC
         ALGORITHM DESIGN. OPERATES ON THE MOTOROLA EDUCATIONAL BOARD.
    25/2/93

                                                                             */

#define SHOW_NET    0
#define SHOW_SENSOR 0

#include <stdio.h>
#include "netless.h"
                /* STANDARD HEADER FILE*/

/****************************************************************************

        TRANSFER OF OUTPUT FROM ONE NODE TO THE INPUT OF THE NEXT IS HANDLED
                AS FOLLOWS:

        THE NODE WHICH IS OUTPUTTING KNOWS WHICH NODE IT IS OUTUTTING TO
                AND IT MULTIPLIES IT'S OUTPUT BY A KNOWN WEIGHT FACTOR. THIS
                NUMBER IS THEN ADDED TO THE INPUT NODES EXISTING INPUT.

        THE NODE RECEIVING THE INPUT HAS A SINGLE INPUT POSITION. THIS NUMBER
                IS CHANGED BY THE INPUT FROM ANOTHER NODE HOWEVER THE INPUT NODE
                DOES NOT KNOW WHICH NODE IS INPUTTING.

****************************************************************************/

/****************************************************************************
        operation of network:

                Numbers inputted are added to the exstant value in the receiving
                nodes input buffer.

                The numbers from node[number].input
                                & from node[number].last are used to evaluate the
                inputsum of the node.

                This inputsum number is then stored in the nodes node[number].last
                location.

                node[num].last figure is then used to calculate the nodes output
                based on the gain characteristic of the nodes.

                After all nodes have calculated and distributed their output the
                value of node[number].input is set equal to node[num].buffer and
                node[number].buffer is set to zero.

                The level one nodes produce an output based on the input without
                buffering. i.e. if a senosry node detects any input then maximum
                frequency output is generated instantly without use of the
                calc_op() and sum_inputs() functions.
****************************************************************************/
int GEN_RATIO=0;
int NUM_NODES=0;
                /* GLOBAL VARIABLES   DEFINITION */

struct neuron
        {
        int buffer;
        int input;
        int last;
        int offset;
        int weight[MAX_NO_OP];
        int nextnode[MAX_NO_OP];
        } node[MAX_NODES];
                /* THE NEURON STRUCTURE IS KNOWN THROUGHOUT THE TEXT AS A NODE    */
                /* EACH STRUCTURE CONTAINS ALL THE INFORMATION PRETAINING TO A    */
                /* SINGLE NODE*/

int intarray[MAX_NODES*MAX_NO_OP*2]={1,2,3,4,5};
                /* ARRAY WHICH STORES ALL THE NETWORK INFORMATION  */

/****************************************************************************

        PEEK: FUNCTION TO READ FROM A SPECIFIC ADDRESS IN THE DEDICATED BOARDS
                MEMORY

****************************************************************************/

byte_t peek(address)
        addr_t address;
                /* THE VARIABLE TYPES byte_t and adr_t ARE DEFINED IN NETLESS.H  */

{
  return(*address);
}

/****************************************************************************

        POKE: FUNCTION TO WRITE  'VALUE' TO A SPECIFIC ADDRESS IN THE
                DEDICATED BOARDS MEMORY.

****************************************************************************/

void poke(value, address)
        byte_t value;
        addr_t address;

{
  *address = value;
}

/****************************************************************************

        INIT_PIT: THIS FUNCTION SETS UP THE PI/T FOR 8 BIT I/O.
                PORT A IS THE INPUT PORT AND  PORT B IS THE OUTPUT PORT.
                THE FUNCTION ALSO INITIALISES THE TIMER ON THE PI/T.

****************************************************************************/

void init_PIT()
{
poke(0x00,CNTRH);               /* INITIALISE COUNTER  VALUES                      */
poke(0x00,CNTRM);               /*             "                              */
poke(0x00,CNTRL);               /*             "                          */
poke(0x01,TSR);            /* INITIALISE TIMER STATUS REGISTER     */
poke(ZERO,PGCR);        /* SETS MODE 0: UNIDIRECTIONAL 8 BIT OP.*/
poke(ZERO,PSRR);        /* DISABLES DMA & EXTERNAL INTERRUPTS   */
poke(HI,PBDDR);         /* SETS PORT B AS OUTPUT                */
poke(ZERO,PADDR);       /* SETS PORT A AS INPUT                 */
poke(0X01,TCR);         /* INITIALISE TIMER CONTROL REGISTER    */
poke(0x00,CPRH);        /* INITIALISE COUNTER RESET VALUE   (HIGH BIT)*/
poke(0X00,CPRM);        /*              "                   (MIDD BIT)*/
poke(0xFA,CPRL);        /*              "                   (LOW BIT)*/
poke(0xA0,PACR);        /* SETS SUBMODE 1X => BIT I/O           */
poke(0xA0,PBCR);        /* SETS SUBMODE 1X => BIT I/O           */
}

/****************************************************************************

        CREATE_NODES: THIS FUNCTION USES THE ARRAY VALUES WHICH ARE DOWNLOADED
                FROM THE PC TO THE DEICATED CONTROL BOARD TO CONSRUCT THE NETWORK

****************************************************************************/
```

```
void create_nodes()
{
        int count;
        int node_num;

for(node_num=0;node_num<MAX_NODES;node_num++)
        {
#if SHOW_NET
        printf("\t%d\n",node_num);
#endif
        node[node_num].offset = 0;
        node[node_num].input  = 0;
        node[node_num].buffer = 0;
        node[node_num].last   = 0;

        if (node_num >= NUM_SENSORS + NUM_MOTORS)
                /* IF NOT AMONG THE EIGHT PERMENANT NODES*/

                {
                if (GEN_RATIO != 0)
                        /* IF THE GENERATOR NODE      RATIO     (WHICH     IS     DETERMINED
*/
                        /* EMBRYONICALLY) IS NONZERO THEN ENTER LOOP*/

                        {
                        if  ((node_num % GEN_RATIO) == 0)
                                {
                                node[node_num].offset = 25500;
                                node[node_num].last   = node[node_num].offset;
                                }
                                /* IF THE NODE NUMBER IS A MULTIPLE OF THE GENERATOR   */
                                /* NODE RATIO THEN ADJUST NODAL OFFSET ACCORDINGLY     */
                                /* I.E FOR CONSTANT MAXIMUM OUTPUT*/

                        }
                }

        for(count=0;count<MAX_NO_OP;count++)
                {
                node[node_num].nextnode[count] =
                                intarray[node_num*MAX_NO_OP*2+count];
                node[node_num].weight[count]  =
                                intarray[node_num*MAX_NO_OP*2+(count+MAX_NO_OP)];
#if SHOW_NET
                printf("\t\t%d\t%d\n",node[node_num].nextnode[count],
                                node[node_num].weight[count]);
#endif

                }
        }
}

/***************************************************************************

        SUM_INPUTS: THIS FUNCTION OPERATES ON THE NODE'S INPUTS AND
                STORED INPUT VALUES TO EVALUATE THE OVERALL INPUT FOR THE CURRENT
                TIME STEP.

                INPUTS ARE TREATED AS CURRENTS (I.E. OUTPUT WEIGHTS ARE TREATED
                AS CONDUCTANCES AND OUTPUT VALUES AS VOLTAGES.)

        INPUT  =    ( CAPACITANCE * LAST_INPUT ) +  ( UPDATE_TIME * NEW_INPUT)
                        ÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ
                                ( CAPACITANCE + (UPDATE_TIME/RESISTANCE) )

***************************************************************************/

void sum_inputs(num)
        int num;
{
        long Q1;
```

```
                /* INPUT CHARGE AT TIME (t)*/

        int  Q0;
                /* INPUT CHARGE AT TIME (t-1)*/

        int  input;
        int  last;
input = node[num].input;
Q0    = node[num].last;

Q1 = (((CAP*Q0)+(UDT_TME*input))/(CAP+(UDT_TME/RES)));

if ( Q1 > 25500)
        node[num].last = 25500 ;
                /* UPPER LIMIT ON INPUT VALUE*/

else
        if ( Q1 < node[num].offset )
                node[num].last = node[num].offset;
                /* NODE MUST HAVE A PREVIOUS OUTPUT WHICH EXCEEDS THE OFFSET */

        else
                node[num].last = (int)Q1;
                        /* IF BOTH CONDITIONS SATISFIED THEN VALUE IS THE No. Q1 */

}

/****************************************************************************

        CALC_OP: THIS FUNCTION EVALUATES THE OUTPUT ACTIVATION LEVEL FOR A
                NODE BASED ON IT'S BUFFERED INPUT VALUE AND GAIN CHARACTERISTIC.

****************************************************************************/

int calc_op(num)
        int num;
{
        int  present_input;
        int     input;
        long outval = 0;

present_input  = node[num].last;
input = node[num].input;

if ( (num==MOTOR_A) || (num==MOTOR_B) )
   {
        outval += 50;
        outval += (input/255) * GAIN ;
        if (outval > 100)
                outval = 100;
        if (outval < 0  )
                outval = 0;
        return((int)outval);
   }
                /* SPECIAL CASE FOR OUTPUT MOTOR CONTROLLING NODES SINCE THEY*/
                /* ARE BIPOLAR MOTORS*/

else if (present_input > THRESHOLD)
        {
        outval = (((present_input/255) * GAIN) + LOW_ACT_LIMIT);
        if (outval > 100)
                outval = 100;
        }
                /* IF NORMALISED (0-100) BUFFERED INPUT VALUE IS GREATER THAN    */
                /* THE THRESHOLD VALUE THEN THE OUTPUT IS GREATER THAN ZERO AND  */
                /* IS LIMITED TO 100 */

     else outval = 0;

return((int)outval);

}
```

```
/****************************************************************************

        OUTPUT: THIS FUNCTION PLACES THE CALCULATED OUTPUT OF THE GIVEN NODE
                IN THE INPUT BUFFER OF THE CONNECTED NODES.

****************************************************************************/

void output(num,outval)
        int num;
        int outval;
{
        int count;
        int outtput;
        int nextnode;
        int weight;

for(count = 0; count < MAX_NO_OP ; count++)
        {
        nextnode = node[num].nextnode[count];
        weight   = node[num].weight[count];
        outtput  = outval * weight;
        node[nextnode].buffer += outtput;
                /* THE CALCULATED OUTPUT VALUE IS WRITTEN TO THE NODES INPUT     */
                /* BUFFER TO ALLOW SYNCHRONISATION OF I/O WITH TIME STEPS        */

        }
}

/****************************************************************************

        UPDATE_1: FUNCTION WHICH OBTAINS INPUTS FROM THE PI/T AND WRITES
                THEM TO THE SENSOR 'NODES'.

****************************************************************************/

void update_lev_1()
{
        int             num=0;
        int             input;
        int             sensor;
        int             output_value;
        static int andmask[8]={1,2,4,8,16,32,64,128};

sensor=peek(PADR);

#if SHOW_SENSOR
        if( printf("\n%s%d\t","sensor : ",sensor) == -1)
                {
                printf("\n\n\t%s","error in printf in update_lev_1");
                }
#endif

for (num=0;num<NUM_SENSORS;num++)
        {
        if(sensor & andmask[num])
                output_value=100;
        else
                output_value=0;

        output(num,output_value);
        }
}

/****************************************************************************

        UPDATE_LEV_2: FUNCTION EXAMINES ARRAY VALUES AND CONTROLS THE
                EVALUATION OF THE OVERALL INTERNAL NETWORK'S OUTPUT.
                (HIDDEN LAYERS)

****************************************************************************/
```

```
void update_lev_2()
{
        int num;
        int outval;

for (num=NUM_SENSORS+NUM_MOTORS ; num < NUM_NODES ; num++)
                /* UPDATE EACH NODE */

     {
         sum_inputs(num);
                /* CALCULATE THE INPUT*/

         outval = calc_op(num);
                /* EVALUATE THE OUTPUT*/

         output(num,outval);
                /* OUTPUT THE VALUE TO THE CONNECTED NODES INPUT BUFFERS*/

     }

}

/***************************************************************************

        UPDATE_LEV_3: UPDATE THE NODES CONCERNED WITH THE ACTIVATION OF MOTORS

***************************************************************************/

void update_lev_3()
{
        int inputsum;
        int output_A,output_B;
        int mot_val;
        int outval;

sum_inputs(MOTOR_A);
outval = calc_op(MOTOR_A);
output_A = (outval*16 / 100);
                /* NORMALISATION*/

if(output_A > 15)
        output_A = 15;
                /* OUTPUT LIMIT ON MOTOR*/

#if SHOW_SENSOR
        printf("%s%d\t","Motor A : ",(output_A-8));
#endif

sum_inputs(MOTOR_B);
outval = calc_op(MOTOR_B);

output_B = (outval*16 / 100);
                /* NORMALISATION*/

if(output_B > 15)
        output_B = 15;
                /* OUTPUT LIMIT ON MOTOR*/

#if SHOW_SENSOR
        printf("%s%d","Motor B : ",(output_B-8));
#endif

output_B = output_B * 16;
mot_val = ((output_A) | (output_B));
                /* COMBINE THE TWO 4 BIT MOTOR VALUES INTO ONE 8 BIT NUMBER    */

poke(mot_val,PBDR);
                /* OUTPUT THE VALUE TO THE PI/T*/

}

/***************************************************************************
```

```
        UPDATE_NET: NETWORK UPDATE CONTROLLER.

******************************************************************************/

void update_net()
{
        int num;
        int input;
        int buffer;
        int ipsum;
                        /* INPUT SUM*/
        int output;

while(1)
        {
        update_lev_1();
                        /* INPUT SENSORS*/

        update_lev_2();
                        /* HIDDEN LAYERS*/

        update_lev_3();
                        /* OUTPUT MOTORS LAYER*/

                /* UPDATE ALL 3 LEVELS (INPUT / HIDDEN / OUTPUT )*/

        for (num=0;num<MAX_NODES;num++)
                {
                node[num].input = node[num].buffer;
                node[num].buffer = 0;
                }
        }
}

/******************************************************************************

        POLL_TSR. ALLOWS TIMING OF UPDATE ON DEDICATED CONTROL BOARD.
                THE UPDATE TIME STEP IS CHANGED BY ALTERING THE INITIALISATION
                ROUTINE OF THE PIT.

******************************************************************************/

void poll_TSR()
{
while ( (peek(TSR)&(0x01)) !=(0X01));
}

/******************************************************************************

                                                MAIN FUNCTION

******************************************************************************/

void main()
{
init_PIT();
create_nodes();
while(1)
                /* CONTINUE LOOPING UNTIL SOFTWARE OR HARDWARE RESET IS APPLIED*/

        {
        update_net();
        poll_TSR();
        }
}
```

```
/***************************************************************************
*
*             STANDARD HEADER FILE FOR USE WITH SBC 68000 BOARD           *
*                                                                         *
***************************************************************************/

typedef unsigned char byte_t;
typedef char *addr_t;

#define PGCR  (0x0E0001)       /* PORT GENERAL CONTROL REGISTER          */
#define PSRR  (0x0E0003)       /* PORT SERVICE REQUEST REGISTER          */
#define PADDR (0x0E0005)       /* PORT A DATA DIRECTION REGISTER         */
#define PBDDR (0x0E0007)       /* PORT B DATA DIRECTION REGISTER         */
#define PIVR  (0x0E000B)       /* PORT INTERRUPT VECTOR REGISTER         */
#define PACR  (0x0E000D)       /* PORT A CONTROL REGISTER                */
#define PBCR  (0x0E000F)       /* PORT B CONTROL REGISTER                */
#define PADR  (0x0E0011)       /* PORT A DATA REGISTER          */
#define PBDR  (0x0E0013)       /* PORT B DATA REGISTER          */
#define TCR   (0x0E0021)       /* TIMER CONTROL REGISTER                 */
#define TIVR  (0x0E0023)       /* TIMER INTERRUPT VECTOR REGISTER        */
#define CPRH  (0x0E0027)       /* COUNTER PRELOAD REGISTER (HIGH)        */
#define CPRM  (0x0E0029)       /* COUNTER PRELOAD REGISTER (MIDDLE)      */
#define CPRL  (0x0E002B)       /* COUNTER PRELOAD REGISTER (LOW)         */
#define CNTRH (0x0E002F)       /* COUNTER REGISTER HIGH                  */
#define CNTRM (0x0E0031)       /* COUNTER REGISTER MIDDLE                */
#define CNTRL (0x0E0033)       /* COUNTER REGISTER LOW                   */
#define TSR   (0x0E0035)       /* TIMER STATUS REGISTER                  */
#define ZERO  (0x00)
#define HI   (0xFF)

#define MAX_NODES   50
#define NUM_SENSORS 6
#define NUM_MOTORS  2
#define MOTOR_A       NUM_SENSORS
#define MOTOR_B       NUM_SENSORS+1
#define LOW_ACT_LIMIT  100
#define THRESHOLD      10000
#define GAIN           1       /* Unity Gain Characteristic              */
#define MAX_NO_OP      10
#define UDT_TME        0.001   /* Network update time                    */
#define RES            5000
#define CAP            0.01/RES
```