Evolving Controllers For Walking Robots

C.J.Bainbridge

27th July 2001

Abstract

This project aimed to create controllers that display a walking behaviour in a physically realistic 3D simulation of a legged robot. Traditional approaches to the development of robot controllers and their corresponding architectures are discussed, along with motivations for exploring new techniques. A number of controllers were created using genetic algorithms based on biological reproduction. This enabled generations of controllers to be gradually refined through the process of evolution, finally producing controllers which could walk successfully.

The controllers were interfaced directly to sensors and actuators in the robot simulation. The physics simulator used is a high quality commercial product. This realistic approach was taken as it allowed better estimates about the performance of these controllers on real robots to be made.

Neural network control architectures were chosen as they are scalable and biologically plausible. Individual neurons operate asynchronously and concurrently, and are autonomous. There is no form of centralised control, which makes neural networks highly resilient to damage. It also makes them more appropriate for evolutionary techniques, as small changes in the network (such as adding neurons and changing connection weights), are likely to produce small, incremental changes in the networks performance.

A variety of different techniques were employed to create controllers. Different evolution strategies were tested, including the SGOCE developmental encoding. Totally interconnected network architectures were evolved, as well as architectures which contain a restricted number of connections. Two control systems for the actuators were implemented. Different neuron models were also tested. The performance of these various competing systems is contrasted and evaluated

Contents

List of Figures

1 Introduction

Creating controllers that enable a robot to walk is a complex problem and an ongoing research topic. By synthesizing biologically based controllers we can learn a lot about how walking is performed by the vertebrate nervous system. Aside from academic interest, enabling robots to walk is a commercially desirable activity for use in many of the so called D^3 - "Dull, Dirty, and Dangerous", areas. Legged robots have the potential to be used when wheeled movement is restricted or impossible, such as in a jungle or over rough terrain. A robot equipped with gripping claws is capable of climbing steep surfaces that would stop a wheeled robot. In many real world situations a legged robot will be more versatile and require less energy than a wheeled counterpart. They could be used in space exploration, military reconnaissance and warfare, in deep sea operations, mining and in the nuclear industry.

Sony and Honda are two companies competing to produce the first generation of successfully walking robots. Both have demonstrated walking bipeds [17, 21], and Sony's Aibo robot pet [20] has been a surprise hit in homes around the world. Although state of the art, the walking biped robots that these companies have produced still rely on vast computing power to control stable walking, and they are unable to recover from positions of instability.

Many approaches have been tried to tackle the problem of dynamic robot walking. Mathematicians have applied the study of inverse kinematics to create walking controllers. Neuro-physiologists have studied the nervous system of humans and animals in an effort to understand how the walking problem has been solved by biological evolution. Both approaches have resulted in the synthesis of control systems capable of walking. In the past decade computers have become powerful enough to apply genetic algorithm techniques to artificially evolve walking controllers. It is this approach which this project will focus on.

1.1 Goal of the Project

The goal of this project is to investigate different evolutionary approaches to synthesizing artificial neural networks for complex tasks. There are several choices of neural architecture and genetic algorithm which lead to different styles of evolved networks. These will be compared by experimentation on a control task of producing walking behaviour in a simulated robot. To enable this to be carried out various software libraries are developed and used for genetic algorithms, and neural/physical simulation. The aim of the simulated environment was to be as realistic as possible to enable good estimates to be made of how well these controllers could perform in the real world.

1.2 Summary of Achievements

A sizable object-oriented library of routines has been developed. These provide functionality to simulate physical realistic articulated bodies inside a given environment, to simulate neural networks with models of different neurons, and to carry out distributed evolution with genetic algorithms. These routines were used to develop a front-end to the software that consisted of a scripting language through which artificial evolution could be easily carried out. Using these tools various types of neural networks were evolved with different genetic algorithms, and their performance compared. Other programs were developed to visualise neural networks and physical simulations, which enabled a better appraisal of the walking behaviour.

2 Background Material

This chapter will cover the research undertaken and knowledge necessary to understand the implementation of this work. Understanding walking motion has been a topic of interest to researchers from various disciplines, such as mathematics, biology, and neuroscience. A brief evaluation of this research is given along with some theories that attempt to explain how walking is actually carried out by biological creatures. We will then cover how biological neural networks operate, and how they can be simulated to create artificial neural networks that display similar properties and behaviours. The building block hypothesis will be explained, and reasons given as to why it's a good theory of evolutionary success. The field of evolutionary robotics will be surveyed, and past attempts at evolving walking controllers will be considered.

Artificial evolution will then be explored in more detail. We will take a look at the different ways of encoding neural network parameters into chromosomes, and schemes for parental selection and offspring reproduction.

2.1 Past Walking Research

Some researchers have tried to take a strictly mathematical approach to walking [1]. They use kinematics, which is the forward application of knowledge about joint angles and orientations to calculate the current position of a 3D articulated body. Inverse kinematics allow the researcher to calculate possible joint angles that will put the model in a given position. The walking process can be divided into a series of desired positions, such as leg up, then leg forward. From these positions the changes in joint angles can be calculated, and forces applied at the actuators to produce these changes. At all times the model is kept stable by ensuring that the joint changes are coordinated to keep the centre of mass acting through the models contact points with the floor.

This approach produces very unrealistic behaviour; the models are capable of walking but their motion is either broken into many small discrete changes, producing a "robotic" movement, or longer but smoother changes, where the model goes through unrealistic sweeping motions with its limbs and body in an attempt to maintain balance. Although the models are capable of some impressive feats, such as a biped sitting down or performing a gymnastic style pirouette, they are incapable of behaviour like running or jumping which would cause them to pass through moments of instability.

Dynamically stable controllers, which do pass through unstable postures, and which utilise inverse kinematics, have also been created [18]. The walking behaviour of these models is also unnatural - it is highly timing dependent, so changing speed or direction is likely to fail, and the controllers are unable to recover once they have lost balance. Many of these controllers work by "falling" through the instability, where it is known in advance that gravity and the morphology of the body will act to move into the next desired pose. These controllers

utilise finite state machines, and are therefore unlikely to resemble the neural networks which control walking in biological lifeforms.

Walking in animals and humans has been studied by physiologists and neuroscientists throughout this century. Their experiments have shown that walking, and other cyclic motions of the body, are reflex actions generated by parts of the nervous system known as "Central Pattern Generators" (CPGs). These consist of collections of neurons which generate rhythmically recurring patterns of output signals. It is these signals which are presented, via motoneurones, to the muscle actuators to drive the body through its walking gait. The dynamics of the CPG cause its neurons to produce activity patterns which oscillate, driving the CPG into further activity, and eventually returning the network to its original state from which the cycle begins again. If the neurons of a CPG lose synchronisation the dynamics of the network ensure that they will fall back into the correct rhythmic pattern.

It has been theorised that every body part that makes cyclic movements is controlled by an individual CPG. Experimental evidence has shown this to be true in the case of the lamprey. Due to the unique biology of the lamprey neurophysiologists have been able to map out all of its CPGs and synthesize their behaviour [7]. Some researchers have even gone so far as removing the brain and central nervous system of the lamprey from its body, and using it to control a robot capable of object avoidance and light following behaviour [8].

In animals different CPGs can be connected so that they can produce a coordinated behaviour. For example, CPGs for different muscles in the same limb could be connected to a limb CPG which would coordinate limb movements. This limb CPG might in turn be connected to other limb CPGs so that together they could produce coordinated behaviours. Different gaits may be accomplished by a CPG with many stable attractor states, with movement between these states happening when some state becomes more stable than the current state. There is evidence for this in cats; when the spine is broken stimulation of nerves below the point of severance will cause the legs to walk, with different frequencies of stimulation producing different walking gaits.

It is likely that humans and other legged vertebrates have networks of CPGs which operate in a similar hierarchical manner. There has been an argument as to whether CPGs are present in humans at all, or whether human motor activity can be explained by learned patterns of behaviour. It has been shown that the parts of the brain which generate CPG motory control functions are similar in all creatures from reptiles to primates [24], and from our knowledge of evolution we can expect that humans will have inherited CPGs from lower lifeforms. There is some anecdotal evidence of CPGs in humans - paralysed people have been known to display rhythmic movements. In one published report from Miami a paralysed person being treated for a hip infection was shown to have generated rhythmic motions of the lower limbs. It may be the case that the tremors of patients suffering from Parkinson's disease are caused by malfunctioning CPGs.

Sensory perception enables the nervous system to detect the current position

of the limbs, the forces being exerted by muscles, and contacts between the body and other objects, and is important in providing timing information to the CPGs. Experiments on deafferented animals, in which the sensory nerves are cut so there is no feedback to the nervous system, have shown that sensory feedback has a modulatory effect. Without feedback the CPGs will still generate similar patterns of activity to drive the muscles, but the timing of body motion and the CPG can become out of phase, and hence walking motion will become erratic.

Although most sensory feedback only has a modulatory effect on the patterns generated by a CPG, certain types of feedback can have a more extreme effect. Forcing the shell of a scallop shut has been shown to completely inhibit activation of its adductor muscle. Similarly, moving the tail of a paralysed dogfish from side to side will stimulate its motoneurones to produce coordinated bursts of activity at the imposed oscillating frequency. Sensory feedback is thus important for reacting to these extreme events. Although it's unlikely that any events of this magnitude will occur in walking behaviour, similar extreme changes in behaviour could occur in response to a sudden input change from a higher level controller, e.g. coming to a halt when running.

Higher level control of the CPGs can come from other parts of the nervous system, such as the cerebellum. There are a lot of interconnections between CPGs and higher level control centres. Researchers have used staged evolution to evolve higher level controllers capable of reacting to environmental stimulus, such as light following, object avoidance, and catching visually perceived falling blocks [9]. In all of these cases walking behaviour was evolved first, followed by more complex behaviours building upon the lower ones.

2.2 Biological Neural Networks

A neural network is a collection of simple processing units known as neurons. They are found in the central nervous system and brains of living creatures, and collectively perform all of the sensing and control behaviour displayed by the creature. At the lowest level of control sophistication lie simple creatures which can only react to their environment through reflex actions. Sensor neurons receive stimulus from the environment, which is then processed by intermediate neurons before being turned into actions by motoneurones which are connected to muscle fibers. These simple static feed-forward networks provide enough processing power to account for the reflex arc present in animals. Cyclic actions such as those displayed by central pattern generators actually require no input signals, relying on the dynamics of the network alone to create repetitive signals. More sophisticated creatures have complex neural networks which utilise feedback and dynamic changing of neuron and interconnection properties. These networks are capable of storing information, and can adapt their processing to better control the creatures body.

Neurons operate asynchronously, as there is no centralised control mechanism to provide timing information for the network. Individual neurons are relatively autonomous in that they have no direct knowledge or access to the states of other neurons - their output depends only on their own state and the signals which they receive from other neurons. Recent experimental evidence has shown that neurons are not completely autonomous since both the behaviour of the molecules which transfer information between neurons and the actual processing performed by the neuron can be modulated by molecules released by other neurons [5]. For most released molecules this simply alters the signal between connected neurons, but the chemical structure of nitric oxide molecules allows them to diffuse isotropically through the brain regardless of intervening cellular structures. These gives a modulatory effect on neurons and connections which are close to the molecules point of release in 3D space, rather than needing to be directly connected via a synapse. This is a radically different behaviour to to the point to point information transfer which neural networks were believed to have.

Neurons are analog processing units, which is unsurprising as they exist in the real (continuous) world. They operate on signals which are continuous in both the activity and temporal dimensions.

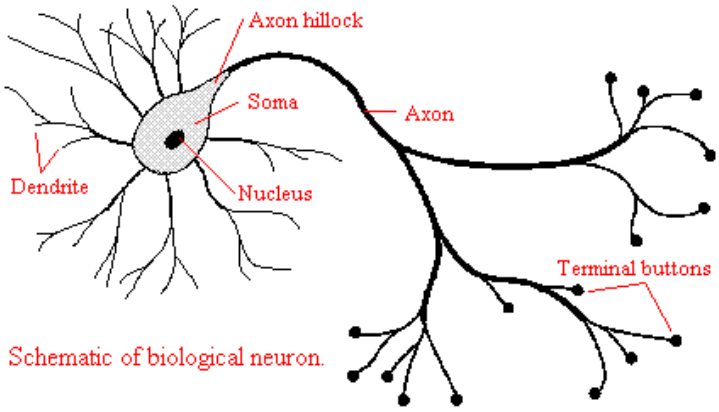


Figure 1: A biological neuron

Figure 1 shows a simple biological neuron [10]. The dendrites are the inputs to the neuron. Signals arrive along them from the outputs of other neurons. The axon carries the output signal from the neuron to terminal buttons. The dendrites present inhibitory or excitatory signals to the neuron which have a cumulative effect in stimulating it. If the sum of these input signals exceeds a threshold the neuron will fire, producing an activity spike along the axon. This signal will arrive at the terminal buttons, which will release neuro-transmitters into a gap (the synapse) between the button and a dendrite of the target neuron. The properties of this synaptic gap determine the strength of the signal

received by the target neuron, and whether it will excite or inhibit its activation. Although it is possible that physical properties of the dendrite and axon connections might perform some processing of carried signals, it is not believed that this is the case as they seem to preserve the original signal as accurately as possible.

Biological neural networks (such as mammalian brains) typically have a huge number of neurons, which are greatly interconnected. Human brains have approximately 100 billion (10^{11}) neurons, of which each is connected to around 1-10 thousand synapse inputs, making a total of around a quadrillion (10^{15}) synapses. Neurons are slow compared to transistors - the fastest neuron switching time is 10^{-3} seconds, whilst transistors can can switch at speeds of upto 10^{-10} seconds [26].

The large number of neurons and connections in biological networks makes them highly resilient to damage - around 1000 neurons die naturally each day of a humans life. There are many cases of people receiving enormous amounts of damage to the brain and showing no changes in their body control or personality. In one unusual case a man was shot in the head from point blank range. After regaining consciousness he showed no ill effects, managing to walk without assistance to the nearest hospital. Other cases have shown that even when the brain is split in half people can still function successfully. The flip side of this is that human behaviour is produced by extremely complex interactions between large numbers of neurons. A small amount of damage will often cause no loss of functionality but the person's personality will be altered.

There are many different types of neurons which can be differentiated by the specifics of the processing that they perform, and by the types and synthesis rates of the primary and secondary neuro-transmitters which their synapses release. Interestingly, stem cell research has shown that neural stem cells can create any type of neural cell found in the body [27]. Understanding how these stem cells organise and structure the formation of the brain may lead to better systems of artificial evolution.

2.3 Artificial Neural Networks (ANNs)

Artificial neural networks are an attempt to synthesise the networks that we see in nature. The principal motives for this are to gain further insight into the function of biological networks and to create something that is actually useful. Commercial applications of neural networks were non-existent until the 1990's, mainly due to lack of affordable computing power. Neural networks have now become the favoured architecture for pattern recognition, leading to a variety of applications in machine vision, fraud detection, speech recognition, and many other tasks which require robust and reliable generalisations to be made from noisy input data.

Research into ANNs was pioneered by Bernard Widrow of Stanford University in the 1950s [26], who discovered the LMS (least mean square) training algorithm.

At the time many great predictions were made about technology and these neural networks which were based on brain function itself! Within 50 years computer intelligence would exceed that of humans, and the greatest dreams of science fiction would be realised. Alas, 50 years on our most optimistic predictions of this are still 50 years away. We now realise that the problems being posed were much harder than we first thought. Even a challenge of beating the world chess champion, a task that seemed ideal for computers, has proved to be a formidable challenge only recently achieved. In sensing and control tasks the ANNs of today are comparable to the most basic insects in their complexity and demonstratable behaviour.

ANNs have been created from many different technologies. The simplest approach, and the one used in this project, is by simulation on a digital computer. This allows us to harness the power of neural networks without having to consider low level physical and mechanical details. A neural network is an interconnected collection of neurons. Each neuron operates with some strictly defined mathematical function which transforms the neurons activity into an output value which can be propagated through its outgoing connections. Connections between neurons are weighted so that the signal is amplified or reduced accordingly. The neuron activity is calculated by computing the weighted sum of the incoming signals, which is the same as computing the dot product of the vector of input signals with the vector of weights.

The function which transforms this input activity value into an output activation defines the neuron's behaviour. In the simplest case the neuron will clip the activity value to its minimum and maximum output boundaries, and just propagate the signal otherwise unaltered. Thresholding the input [28] to produce an "all or nothing" output was common until training techniques that relied on computing the gradient of the output were discovered. A sigmoid activation function produces an output clipped to the boundaries (usually between 0 and 1), and with an "S" shaped curve whose gradient can be calculated. These produce simple behaviours from the neuron; its output will increase in response to increased activity. More complex behaviours are achievable using neurons which have an internal state, where the input affects its 1st, 2nd or higher order change. Neurons with higher order functions are capable of producing complex waveforms such as oscillations and periodic functions.

The values carried between neurons in most simulations are not intended to have the same meaning as those in biology. Biological neural activity is characterised by rapid spikes as a neuron fires. Information is coded temporally in these spikes, so the frequency of the spiking becomes important. For an analog system in the real continuous time world this doesn't represent a problem, but for digital simulations time and the reaction of neurons are not continuous and must be divided into suitable small periods. Simulations therefore tend to code the signals between neurons as the frequency of the spikes. This preserves the temporal coding of biological systems at the expense of the amplitude coding. It is not clear if viewing the information coding differently in ANNs is benefi-

cial, especially in an evolved network where there is no predefined concept of information transfer.

For the neural network to perform some useful function the connection and neuron parameters have to be set. For threshold neurons this means setting a suitable threshold for each individual neuron. For more complex models other parameters such as the adaptation rate of the neuron to its input signal have to be set. For connections the weight of each connection between two neurons has to be set. Much research has been done into finding ways of computing these values when training data is available. The most successful systems employ progressive evaluations and adjustments to the network, eventually stopping when the network performs well on the given task. The main problem in training like this is one of credit assignment - when an incorrect output is observed from the network, how do we adjust the thresholds and weights of all the neurons to correct it?

Backpropagation has been the most widely used training method to date. When an incorrect output is observed each neurons contribution to the output is estimated. This depends on two things: the output of the neuron, and its depth in the network. If a neurons output is 0 then its neutral and not affecting the incorrect network output, but if its 1 then it will be affecting the output strongly. The gradient of the neuron's activation function is then used to calculate how much the outgoing connection weight should be adjusted by, weighted by the neurons estimated significance and a global learning rate. Sigmoid neurons are usually used as they have an easily computable gradient. Adjustments to the weights of all connections are calculated over a set of training patterns, and then averaged and applied to the network. Training then continues on other examples of input/output patterns.

There are a few problems with backpropagation that motivate the search for better algorithms. It is not clear how to create the network topology. Simple choices such as how many neurons should there be, and the degree of interconnectedness are made by intuition, trial and error. Backpropagation is difficult on networks which contain loops and feedback. How can the significance of a neuron's contribution to the networks output by estimated when it can excite and inhibit its predecessor neurons, which in turn excite it in a continuous loop? Unrolling loops is one unsatisfactory approach that misses the cycling activity of feedback. Feedback is an essential element of control theory, and is likely to play a large part in any neural network of significant worth. Totally interconnected networks, where every neuron is connected to every other neuron, are the most expressive and powerful, and yet backpropagation is totally unsuitable for training these networks. Another problem with backpropagation is that it limits the range of behaviour of which the neurons are capable since the activity function has to be differentiable.

In many ways artificial neural networks are similar to the biological ones which inspired them. Neurons act in a way which models their biological counterparts, providing a function with an arbitrary number of inputs and a single output.

They operate asynchronously, and higher order ones are capable of complex behaviours like those displayed by biological neurons. The synaptic potential which alters the strength of a signal passing between a terminal button and a dendrite has the same effect as multiplying the signal by a connection weight. Synapses can also invert the polarity of a signal, as can multiplication by a positive or negative weight. Biological networks tend to be on a much larger scale than artificial ones, containing more neurons and more interconnections between them. This can be seen as a limitation more of our design skills than of technology, as modern processors are quite capable of simulating networks with many millions of neurons and connections in real-time. Simulations on digital computers are inherently digital but it is not clear that this makes any difference. The quantisation of temporal and electrical continuity approximates reality in that a continuous system allows states to become so similar they can't be separated. Since neurons (and all other computing elements) categorise input signals into different states as part of their function continuity is probably not necessary.

2.4 Neuron Models in Walking Research

The activity of a neural network depends on the processing performed by individual neurons. Each neuron computes its current output value using some function which can depend on its weighted inputs, its internal bias, and its state defined by its previous activity. In past robot walking experiments many different models have been used [3]. The sigmoid neuron is one of the simplest models. The weighted inputs are summed, and then this value is entered into a sigmoid function to compute the output.

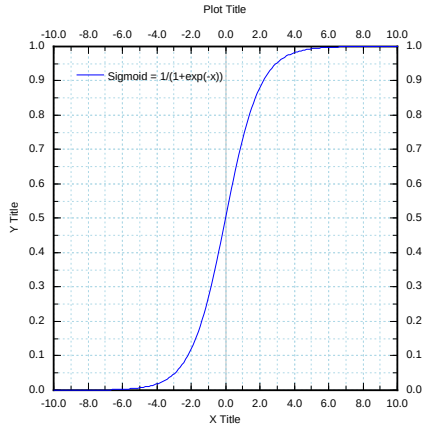
The activity function is:

$$y = -t + \sum_{j=1}^{n} W_j I_j$$

The output function is:

$$O = \frac{1}{1 + e^{-y}}$$

The output of this function relies purely on the current inputs, hence it has no state or continuity dynamics. Despite this it has been used to evolve networks capable of walking in multi-legged robots. The graph of this function is:



Another model, known as "continuous time recurrent neural networks", or CTRNNs, was originally created by Beer for robot walking. The activation function produces the gradient of the change in activity between the previous activation of the neuron and the current inputs, biased by an adaption rate (or "time constant"). The gradient and the previous activity can then be used to calculate the current activity, which is summed with a bias value and placed into the sigmoid function to calculate the new output value of the neuron.

Activity function:

$$a\frac{dy}{dt} = -y + \sum_{j=1}^{n} W_j I_j$$

where a is the adaption rate (time constant).

Output function:

$$O = \frac{1}{1 + e^{b-y}}$$

where b is the bias value.

This model produces continuous dynamical behaviour based on the first order differential equation. Other than that its basically the same as a sigmoidal neuron. This model has been used successfully for robot walking [2][**?**].

A more complicated model has been used for 2D robot walking by Taga. It uses two coupled first order differential equations to produce behaviour like a second order differential equation. Its neuron model is capable of more complex behaviours like oscillation [3].

A third order differential model has been used by Wallen to model the CPGs in a lamprey [22]. These neurons are capable of even more complex behaviour, and are claimed to be more biologically plausible as they were designed to recreate the dynamics observed in the lamprey brain.

2.5 Genetic Evolution

Evolution is the process by which superior individuals in a species survive and reproduce, whilst less successful individuals die. It has long been observed that children inherit features from both of their parents, and this applies equally to plants and animals. Since the offspring of the superior individuals inherit features from them, they are likely to inherit the characteristics that made their parents fit enough to survive and reproduce. These features are in turn passed on to their own children. Through the reproduction of successful individuals to the detriment of unfit individuals good traits are passed through generations and will eventually come to be distributed throughout the population.

Genetic evolution is credited with producing the diversity and complexity of all life on this planet. Over a time span of roughly 3.5 billion years life has evolved from single cell organisms to modern day plants and animals. Although it has taken a long time evolution has proved to be an incredibly powerful technique - the existence of beings as adaptable and intelligent as humans is testament to this.

Humans have been using evolution to grow plants and animals with desirable characteristics for a long time; in fact, it has been known for thousands of years that individual characteristics are passed from parents to their children. This has been used to enhance characteristics such as speed (in horses), disease resistance (in wheat), amount of fruit borne (in trees), colour (of flowers) and appearance (of certain canine breeds). Some of these human directed evolutions may have occurred naturally, but things such as the breeding of dogs to accentuate physically appealing features (e.g. large heads in the British mastiff bulldog) have created an unnatural evolutionary path leading to a population which would be unable to survive in the wild. In the case of pedigree bulldogs the heads of newborn pups are now too big for natural birth, so they all have to be delivered by Cesarean. This may seem foolish, but it is also displays the power of evolution well - the species has been able to adapt and survive in a much changed environment even when faced with unnatural evolutionary pressures, and future generations will inherit the properties which helped them survive.

In the 1800's an Augustine monk named Mendel discovered that independent characteristics such as height and smoothness of seed would be reliably inherited by the offspring of pea plants. He hypothesised that certain "atoms of inheritance" were dominant, and his experimental evidence with pea plants supported this. For example, Mendel found that a tall pea plant and a short pea plant would always produce tall offspring, suggesting that tallness was a dominant "atom of inheritance". Although he was knew nothing of genetics, Mendel was proposing the existence of what we now know as genes [29].

The cells of all animal and plant life contain a genetic sequence of deoxyribonucleic acid (DNA). This sequence contains the genetic code from which every cell in the body is constructed. DNA is a string of nucleotides; these are simple

structures containing a base, a sugar, and a phosphate molecule that is shared between adjacent nucleotides in the DNA. There are four bases in DNA - adenine, cytosine, guanine, and thymine, known as A C G and T. A and T bond together, as do C and G. When two strands of DNA contain complementary bases they will bond together, and molecular forces will cause them to twist into the familiar double helix. A chromosome is a large strand of DNA; humans usually have 23 pairs of chromosomes.

A sequence of three consecutive bases in a chromosome is known as a codon. When the chromosome is decoded each codon produces either nothing, or one of 20 amino acids. Since there are four bases and $4^3 = 64$ there is some redundancy, which acts as a defence against mutation. Two of the codon sequences act as start and stop signals for the production of a protein. The codons between these are decoded to make a string of amino acids, which are bound together in the same sequence as in the DNA to make a protein. The string of codons between a start and stop codon which contains the instructions for the sequence of amino acids needed to make a protein is known as a gene.

Ultimately genes are responsible for the physical structure of our bodies. They regulate a complex process of chemical interactions that produces a stable, self sustaining bio-ecology. It is this success, the complex behaviour built from small evolved components, which we hope to emulate. It is important to remember that biological evolution is an undirected process which works because of survival and reproduction of the fittest. There was no initial design plan to produce morphologies or controllers, and yet the chaotic nature of the world produces a process that is self organising and would appear, to an external observer, to be directed by intelligent behaviour.

There are two distinct processes at work in biological evolution. When two individuals reproduce their chromosomes are combined in a process known as 'crossover'. To produce a child chromosome the two parent chromosomes both split into two strings. Two of these sub-strings (one from each parent) then join to create the child chromosome. Due to the way that the chromosomes split and the genes bond the child will be of similar length to the parents. This process of crossover allows children to inherit genes from both parents. Asexual reproduction, where there is only one parent, does exist in nature but the majority of advanced species produce their offspring from two parents. This seems to be the optimal number as there are no species who reproduction requires three or more parents. Interestingly, recent advances i n genetic engineering have indeed produced human children with DNA from 3 parents.

The second process at work in biological evolution is mutation. The physical process of copying DNA is imperfect and subject to errors. As previously mentioned the coding of codons has some redundancy which reduces the impact of mutations. Despite this, many mutations will result in codons that manufacture different amino acids and thus change the effects of the gene. In this way mutation introduces new genetic material into the chromosome. The changes may result in a chromosome that performs better, worse, or the same. The

evolutionary process ensures that changes for the better will survive and be propagated through the generations. Mutation is the only evolutionary force in asexual reproduction. A species cannot evolve without the new genes produced by mutation.

Biological evolution is an incredibly powerful process. It is easy to see how over a long period of time it could produce species that are very advanced. What is not so obvious is that evolution works not only on the physical bodies of the creatures, but also on the way that they evolve - in a way evolution itself is an evolved process. The way in which gene sequences are converted into physical actions, and the way that chromosomes split and recombine in reproduction, are both products of evolution. To simulate all of this would be an impossible challenge - nothing could be taken for granted, and you would have to rely on randomness to discover that cells can contain DNA, that they can split into two under the right circumstances, and that strings of DNA can be processed in a meaningful way. Any attempt to harness this power must be done with the recognition that biological evolution happened over billions of years, and had access to an unimaginable amount of processing power in the form of real world interactions between many, many molecules.

2.6 Artificial Evolution

The study of genetic algorithms is an attempt to transfer the success of biological evolution into the domain of computer based algorithms and software. Genetic algorithms were first investigated by Holland in 1975 [4]. The canonical genetic algorithm developed by Holland has remained unchanged since then, with most of the following research focused on applying the genetic algorithm as an optimisation technique for various problems.

The GA approach is simple - define a mapping between bit-strings (the chromosomes) and potential solutions, and then, starting from an initial population of chromosomes, evaluate each one and combine the best probabilistically to produce the next generation. Hopefully combining good solutions will lead to better solutions, which will in turn be propagated through the population as happens in biological evolution. Eventually all of the members of a population will converge to a point where they have much genetic material in common and the solutions they produce are very close to each other in the solution space.

The genetic algorithm technique borrows the ideas of incremental search and building towards a good solution from artificial evolution. Typically the problem that is to be solved has a large search space; a problem requiring a solution smaller than 30 bits can usually be solved by enumerating the search space. In the problem there will be an explicit function that has to be optimised, known as the fitness function. This is different from the real world, and artificial life research, where the function is implicit survival and reproduction. GAs are good for finding or fine tuning parameters which interact in complex and nonlinear ways.

The fitness function can be any evaluation of the solution which judges how good it is. It is isolated from the search, which differentiates it from techniques such as gradient directed search. The fitness function doesn't have to be able to determine anything about the solution space surrounding a given point. This "black box" treatment of the fitness function makes genetic algorithms suitable for many problems where traditional search techniques would be inappropriate. It also introduces new problems, however, as problem specific information which could accelerate the search is ignored. GAs are only suitable for problems which have many potential solutions with a wide range of possible fitness values. A problem like searching for a cryptographic key would be unsuitable as the fitness surface would be completely flat apart from the single point solution which is correct. The evolution of neural networks is an appropriate problem for a GA since similar networks will display a similar behaviour.

The parameters of possible solutions to the problem are encoded as chromosomes which are then evolved, rather than varying possible values for the parameters separately. An encoding scheme is defined between the genotype, which is the potential solution represented as a chromosome, and the phenotype, which is the solution itself. This encoding scheme can be direct and implicit, like placing the parameters to the fitness function in a binary string, or it can be more complex, like treating a binary string as instructions to "grow" the phenotype. It is essential for high fitness chromosomes to have genetic similarities, as it is this property of the coding which guides the GA search. Manipulation of the chromosome by either crossover or mutation creates new chromosomes which inherit genetic similarities from their parents. The encoding must preserve these genotype similarities when they are mapped to phenotypes.

The initial population will usually consist of random bit-strings. It may be seeded with potential solutions that have already been developed in an attempt to optimise them, or to "cross pollinate" this evolutionary run with genetic information from other populations. Sometimes problem specific knowledge can be applied to create an initial population of reasonable fitness. The chromosomes in the population are then evaluated using the fitness function. Chromosomes with a high fitness are mated by either combining their genetic information to produce a new chromosome, or by mutating them.

The method most commonly used to combine genetic information is crossover. A point in the chromosome is randomly chosen as the breakage point. Information preceding this point is taken from one parent, and the information following it from the other. These two semi-chromosomes are then concatenated to create the new child chromosome. Mutation is used to introduce new genetic material into a population. As mutations are often disastrous, and they can destroy inherited characteristics, a low mutation rate is usually chosen. Goldberg [4] suggests a rate of 1 mutation per 1000 bits. A single bit is mutated by inverting it.

In some applications of GAs (in particular neural networks) crossover has been abandoned and mutation used as the primary reproduction operator. This is

usually done as there is no clear way of combining data from separate chromosomes to produce offspring that are likely to survive. In the case of neural networks creating a child by splicing two different networks together is unlikely to succeed - unless the two parent networks are very similar the changes introduced by combining them will be the equivalent of massively mutating one of the networks, which is clearly undesirable.

It is possible to introduce other genetic operators that manipulate structures inside the genotype using problem specific knowledge. Doing so risks turning the GA into a generalised search that may not produce the desired GA search behaviour, and hence may neglect sampling of large parts of the solution space. In a similar vein the solutions don't have to represented as binary strings, they could be graphs or use some other problem specific representation. Holland argues that in order to achieve the benefits of a GA search all solutions should be coded as binary strings, however it is clear that other codings along with corresponding operations can be equivalent to the manipulation of bit-strings. The argument is somewhat irrelevant since problem specific knowledge is being used to devise a genotype to phenotype mapping that preserves similarities between both, and ultimately everything has to be represented as a bit sequence inside the computer used to carry out evolutionary runs.

Genetic algorithms produce a "beam" search behaviour. Initially solutions are randomly distributed throughout the solution space. Many of these solutions will perform very poorly and be quickly removed from the population. The others will continue the search from many parallel points which will begin to converge. In the total search space any solutions that have a slightly good fitness are likely to be closeby; as an example from the real world consider that humans share 85% of their DNA with the common earthworm, and 98% with apes. The search proceeds in clusters around these points, and in new points created by combining them. As new generations are created the search can be visualised as a beam moving through the search space.

The search is terminated when the members of a population have converged around a common point in the solution space. The individual with the highest fitness is then chosen as the solution. There can be problems - if the population converges too quickly the solution space may not have been searched thoroughly enough and areas of high fitness may have been missed. The population may not converge at all if this problem is inappropriate for a GA, the search space is too big, or a bad genotype representation has been chosen. If the fitness surface contains multiple peaks the population may become distributed between them; different groups within the population will have converged to different solution clusters. There is always a trade-off between exploration and exploitation of knowledge about the fitness surface. Whilst it is desirable for an evolutionary run to converge quickly, speed has to be scarified in order to be thorough and evaluate a diverse enough group of solutions.

2.7 The Building Block Hypothesis

Evolution is successful because smaller parts of a chromosome can be combined to produce bigger parts which preserve the functionality of their components. As populations evolve useful genes become widespread, whilst bad ones tend to die out. These genes can be seen as building blocks, which are joined together to create sets of high performance genes, which can in turn be used as larger building blocks. This hierarchical decomposition of a chromosome into building blocks is a simple and yet effective strategy for analysing an evolving population. Sets of genes which interact strongly in a beneficial way will tend to clump together inside the chromosome. The position of a gene inside the chromosome is irrelevant, since it will act in the same way regardless. However, the reproduction process of crossover destroys relationships between distant interacting genes as the crossover point is more likely to be between them. Offspring won't contain all of the good interacting genes, so they will perform poorly and won't be selected for reproduction. The chromosomes which have the interacting genes together in a close sequence are less likely to have them broken up by crossover, and so their children are more likely to perform well and be themselves chosen for reproduction. This causes the propagation of short sequences of good genes throughout the population.

The building block hypothesis [30] also helps to illustrate the type of problems that evolution is good at solving. Problems which can be broken down into a hierarchy of smaller components, which can then be incrementally solved and combined to create solutions for the larger problems, are particularly appropriate for a genetic algorithm.

2.8 Evolutionary Robotics and Walking

For walking tasks researchers have tended to use networks with a fixed number of neurons which are totally interconnected. The connection weights are then evolved.

In Richard Reeve's PhD thesis [3] he investigated evolving weights in a fixed architecture network (only the weights vary), and in a variable architecture network (the weights and existence of connections vary, and the number of neurons may also vary) with various constraints and found that a fully interconnected fixed network performed best as a walking robot controller. Reeve's controllers used a fixed number of neurons calculated by analysing the structure of the robot, typical controllers consisted of 50-100 neurons.

Reil & Husbands [2]used a network of 10 fully interconnected neurons to control a MathEngine simulated biped. Connection weights, neuron time constants and biases were evolved to produce successful walking behaviour.

For walking controllers the architecture can be directed towards the task rather than using one global controller. Some researchers have experimented with

evolving a separate network for each limb (e.g. Sims [12]), with the networks interlinking through a few connections or with a central controller. Others have tried to evolve one network for the limb type, and then replicating this network for the other limbs and growing connections between them [3]. This is done in the belief that controllers for identical limbs should be similar, and this will ensure that any useful controller found will be used for other identical limbs rather than having limbs that are unused. Reeve reported that building this symmetry into the controllers had a dramatic positive effect on their performance, ensuring that all of the limbs were used, and enabling the limb controllers to coordinate better to produce more stable walking. A similar scheme of reproducing a network for each limb and reflecting them for opposite limbs was used by Kodjabachian and Meyer in [11]. A program was evolved which was interpreted for each cell in parallel to develop the network.

'GasNets' [5] are a type of network proposed by Husbands and colleagues based on observations of the modulatory effects of diffusing nitric oxide in biological neural networks. They have been evolved to perform robot walking in a goal directed manner (locating and approaching a triangle or square in a walking robot). In this network each neuron is given a 2D coordinate which is used to calculate how much nitrous oxide it is exposed to as the gas diffuses in a semi-circle from other neurons. The results reported were encouraging - networks were evolved in 1/10th of the time taken when using traditional neural networks. If nearby neurons process similar information then GasNets are a good model as they allow close neurons to be modulated in the same way. GasNet neurons are more expressive as their function can be altered as the network is running. This enables individual neurons to display a dynamically varying range of behaviours.

An evolutionary approach was used to evolve the gait of Sony's AIBO robot quadruped [19]. In this work rather than evolve a complete control system for walking, parameters of a traditonal walking controller such as raised leg angles, step size, and interleg synchronisation timings were optimised. A real robot was used to evaluate the performance of the controller over a variety of real world surfaces including carpets, rubble, and low friction tiles. This is interesting as it is the first case of genetic algorithms being used to evolve parameters for walking in a consumer oriented robot.

2.9 Chromosome Encoding Schemes

Genetic algorithms operate on chromosomes, which are abstract representations of the data structure being evolved - in this case neural networks. In biological terms, the chromosome is the genotype, and the neural network is the phenotype. The encoding scheme defines how a chromosome can be transformed into a neural network. The chromosome has to contain a representation of all of the parameters which are being evolved, so different encoding schemes are necessary when we are evolving different types of networks. We may wish to evolve only

the connection weights, or we may wish to evolve other parameters such as the topology, neuron types etc.

A good encoding scheme has certain desirable properties. Small changes to the chromosome should produce small changes in the neural network. This allows the process of crossover to work, as the reproduction operators disrupt the genotype, and yet the child phenotype is still similar to its parents. It is also desirable for mutation to produce small changes to the genotype as drastic changes to a phenotype are more likely to destroy the good genetic data from the parents than produce the localised search around them that we desire. The encoding should be highly compact, allowing a complex and detailed phenotype to arise from a simple genotype. One way of doing this is to allow repeated sequences of phenotype to be compressed in the genotype using a sort of run-length encoding. This technique was used successfully in Karl Sims creatures [12] to evolve controllers and 3D morphology.

There are two encoding schemes - direct, where the network weights and perhaps topology are explicitly represented in the chromosome, and indirect, where the phenotype is generated in some way from the information in the chromosome.

Example of direct encoding:

```
Chromosome: W1 W2 W3 W4 W5
```

The chromosome consists of a string of weights. The network topology and connections are already known.

Reil & Husbands [2] used a direct encoding of weights, time constant, and bias for continuous time recurrent neural networks capable of biped walking:

```
Chromosome: W1 W2 ... Wn*n T1 ... Tn B1 ... Bn
```

Each value is a real number, which is clipped into a prespecified range of [16,16] for the weights, [0.5,5] for the time constant, and [-4,4] for the biases. The network topology was already known as the neurons were totally interconnected.

An indirect encoding will typically contain instructions which are processed to create the phenotype. These instructions may be executed in parallel, which is more biologically realistic. They may relate to the development of a network inside some kind of geometric constraints, as in the SGOCE encoding, or they may relate only to the network topology, as in most other research.

Example of indirect encoding:

```
Chromosome: M1 C31 D5
```

The chromosome consists of a string of instructions (e.g. Make neuron 1, Connect neuron 3 to neuron 1, Divide neuron 5).

2.10 SGOCE Developmental Encoding

SGOCE, which stands for "Simple Geometry Oriented Cellular Encoding", is an encoding proposed by Kodjabachian and Meyer [11]. In this system a neural network is developed on a 2D surface. Initially cells (neurons) are placed at certain points on the surface. A program is then run on each cell which can contain several instructions:

DIVIDE a,r.

A copy of this cell is created at an angle a and distance r from the current position.

GROW a,r,w.

An outgoing connection is created with its target as the closest cell to the point at angle a and distance r from the current position. The weight of this connection is w. GROW a,r,w

DRAW a,r,w.

An incoming connection is created with its source as the closest cell to the point at angle a and distance r from the current position. The weight of this connection is w.

SETBIAS b.

The bias of the neuron is set to b.

SETTAU r.

The time constant of the neuron is set to r.

DIE.

The neuron is removed from the network and deleted. This is the end of the program.

Several other commands are available which satisfy grammar constraints but otherwise do no processing:

NOLINK This can be used in place of GROW and DRAW

DEFBIAS Used in place of SETBIAS and indicates that the default bias is being used.

DEFTAU Used in place of SETTAU, indicates default time constant is being used.

SIMULT3 A 3 way split in the program tree

SIMULT4 A 4 way split in the program tree

To reduce the size and complexity of evolved programs they are subject to a strict grammar called GRAM-1. The production rules are:

Start1 = DIVIDE (Level1,Level1)

Level1 = DIVIDE (Level2,Level2)

Level2 = DIVIDE (Neuron,Neuron)

Neuron = SIMULT3 (Bias,Tau,Connex) | DIE

Bias = SETBIAS | DEFBIAS

Tau = SETTAU | DEFTAU

Connex = SIMULT4 (Link,Link,Link,Link)

Link = GROW | DRAW | NOLINK

This decomposition of the grammar is depicted graphically in figure 2.

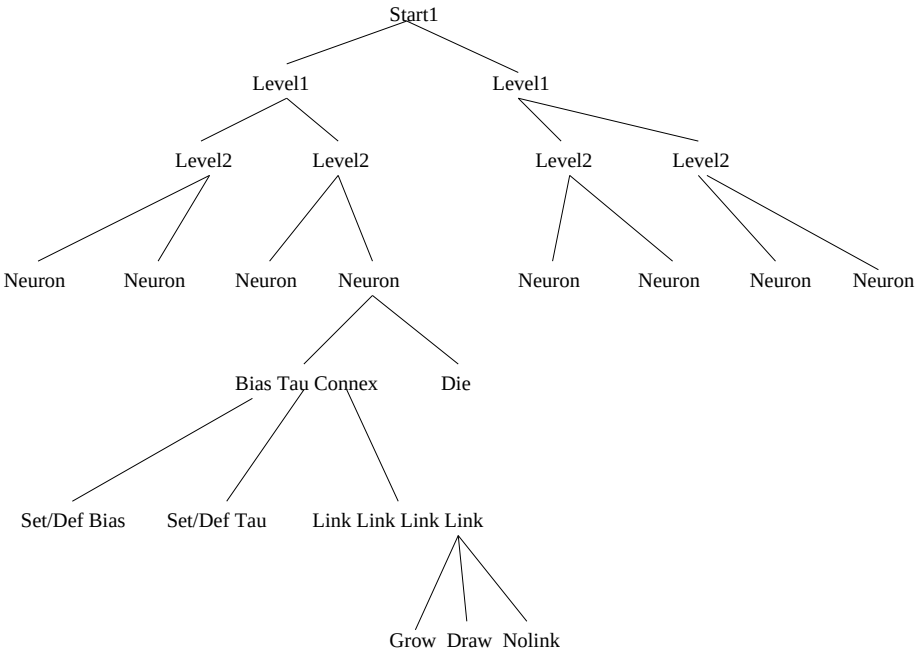


Figure 2: Decomposition of SGOCE GRAM-1 grammar

2.11 Selecting Parents

Parents with a high fitness are chosen to reproduce in the hope that their children will perform as well, or even better, than they did. There is a tradeoff between exploring the solution space, and exploiting the knowledge that we already have. In a typical population we want to choose parents that are likely to produce children that are well performing but also genetically diverse.

The evolution process can be viewed as a movement of clustered points over a fitness surface. Initially the points are randomly distributed over the surface. Due to the nature of the problem most of the surface will be relatively flat and of low fitness. As new generations are produced clusters of points will develop around hills on the surface, and they will begin to ascend them. The individuals on the flat, low fitness plane will die off. We desire the clusters to be tightly focused so that they can climb the hills, and yet at the same time widespread enough that they don't get stuck in local maxima. The evolution should be able to leap over lower fitness areas of the surface to migrate between nearby peaks. When a flat plane is encountered the population should spread out over it searching for paths to higher fitness areas.

High fitness solutions that don't converge are desirable as it indicates that the population has discovered a large plateau in the fitness surface. Children can be very different from each other and still perform well. There is evidence that this is happening in the case of humans - almost all of our children are of an incredibly high fitness compared to other animals, and yet our genetic material is also changing faster than any other species [31].

There are a few popular methods for selecting parents [29]. In roulette-wheel selection chromosomes are chosen with a probability proportional to their fitness. This is useful when the relationship between fitness values and performance is approximately linear. For a chromosome of fitness f_i the probability of being selected in a population of size n is $\frac{f_i}{\sum_{i=1}^{n} f_i}$.

Tournament selection is when a group of a certain size is chosen randomly from the population. The fittest of the group is chosen to reproduce. Tournament based selection is useful for slowing the rate of convergence. The probability of being selected for a population of size n and tournament of size t is $\frac{1}{nt}$.

In rank based selection the chromosomes are ranked according to their fitness. The probability of being chosen is proportional to the rank rather than the fitness value. Rank based selection is useful where there is a dramatic difference between the fitness values for similar chromosomes. For a population of size n and chromosome of rank i the probability of being chosen is $\frac{n+1-i}{\sum_{j=1}^{n} j} = \frac{2(n+1-i)}{n(n-1)}$.

2.12 Reproduction

In neural network evolution creating the child network is almost always done by mutating the parent. Parameter values, such as connection weights, and

neuron time constants and biases are floating point values which are mutated by replacing them with either a randomly generated value, or one drawn from a Gaussian distribution centred on the current value. Replacing with randomly generated values produces a wider search through the solution space which will take more time to improve a chromosome that lies on an ascending ridge along the fitness surface, as most large steps through the solution space will result in chromosomes with decreased performance. On the other hand, replacement with values randomly chosen from the Gaussian distribution means children will lie closer to their parents in the solution space and are hence more likely to lie closely on the fitness surface, which produces a narrower, more focused search.

SGOCE programs are reproduced by first selecting a parent and making a perfect copy of it as the child. A sub-tree of the child is then randomly chosen and replaced by a grammar compatible sub-tree taken from a 2nd parent. This recombination is done with a probability Pc. Next comes the mutation stage, which is carried out with a probability of Pm. A sub-tree is randomly selected and replaced with a grammar compatible randomly generated sub-tree. A second mutation stage is now carried out with probability 1. A certain number of program parameters (where the exact number is drawn from a binomial distribution) are randomly selected and changed slightly. This is the final step in the creation of the child, so it is now ready to be added to the population. In [11] values of Pc=0.6, Pm=0.2, and for the binomial B(n,p) n=6 and p=0.5 are used.

2.13 Summary

This chapter has covered the background knowledge needed to understand how we can use genetic algorithms to create neural networks, why we would want to, and why it works.

3 Implementation

The implementation of this project consisted of the mechanical design of the robot, and programming of the software simulation libraries. The mechanical implementation of the robot covers such design choices as the number of legs, distribution of body weight, and placement of motors. The software was divided into distinct classes which provided functionality for seperate components of the application. The robot simulation describes how the mechanical model was tranferred to a physically accurate computer simualtion. A neural network simulator was written to carry out simulation of the evolved networks. Three distinct classes of genetic algorithms were implemented, and a distributed processing component was written to take advantage of the large amounts of computational power available in a computer network. A script interpreter for the Python programming language was integrated to create a powerful and versatile front-end to the application. Other programs were developed to aid in visualising the neural networks and the physical simulations. In many of these components external libraries are used for functions like physics, graphing, networking and MPEG encoding.

3.1 Mechanical Implementation

The initial design was for a very simple hexaped with a limited range of limb movements that enable it to walk. The upper legs are connected to the main body via hinge joints directed along the horizontal axis. These hip joints have a motor attached which allows forces to be exerted to lift the leg. The orientation of the leg is limited to $\pm 45^o$ from the body plane. This allows the leg to be lifted up and lowered down by the same amount. The upper leg is attached to the lower leg by another motor powered hinge joint. This time the hinge axis is vertical, allowing the lower leg to move forwards and backwards. Again the hinge is constrained so that the intersection angle of the upper and lower leg will not be greater than 45^o. The hinge joints on the hips and knees each have a single degree of freedom. The robot is therefore capable of moving its feet through 2 degrees of freedom. The mass of the body is 1kg, and the upper and lower legs are all 0.25kg.

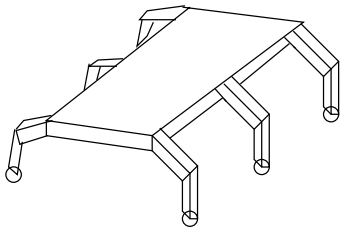


Figure 3: 3D view of the robot

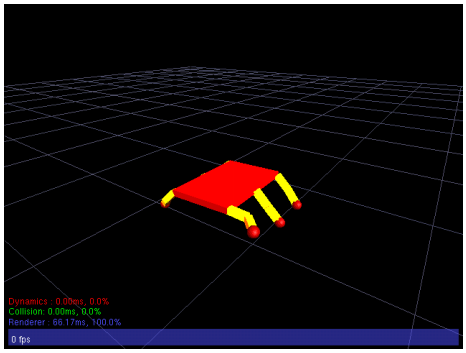


Figure 4: Screenshot from the simulator

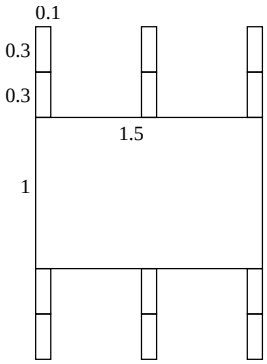


Figure 5: Robot overhead

3.2 Software Implementation

All of the software implementation was written in C++. C++ was chosen as it produces efficient executable code, and the evolutionary process was going to be computationally intensive. In particular, the physics simulation required a lot of CPU time, and used an existing C library (MathEngine). More C libraries were required to simulate neural networks (SNNS), and for distributed computation (PVM). Using C++ was the obvious choice for an object-oriented medium scale software application that was fast and linked to external C libraries.

The software implementation is divided into 5 main base classes - Simulator, Network, Evolver, FitnessTester, and Pvm. These classes are instantiated and then linked together by the calling application, which can be either the main() function of an executable, or a Python script. Each class contains the state and methods for a specific object. The 'Simulator' class contains everything to do with the robot simulation. Network is a class that represents neural networks, and has methods for loading, saving and simulating them. Evolver is an abstract base class that is inherited by any class that performs evolution. An evolver is specific to a certain type of network and evolutionary process. They must provide methods for creating an initial population, loading and saving their populations and state, and running the evolution for a given number of generations. The FitnessTester class performs all simulations on behalf of the Evolver. It can be set to use a distributed computation network by linking to an instance of the Pvm class, or it can run simulations directly when it's linked to an instance of the Simulator class. Classes that require linking to other classes provide methods which allow this to be done at runtime (e.g. FitnessTester.setUnitEvaluate(Simulator s)), which is much more powerful and versatile than static linking at compilation time.

All of the source code was fully documented using the Javadoc syntax. Doxygen [35] was used to produce a 122 page programmers reference manual detailing all of the public interfaces to the C++ classes. This documentation can also be used as a reference for the Python interpreter as the syntax is almost identical.

3.3 Robot Simulator

A commercial physics software development kit was used to create the robot simulation. After researching the available software it was decided that MathEngine would provide the most physically accurate simulation, and enable the quickest software development due to the many examples provided with source code and the quality of documentation. It was also known that MathEngine had been used previously to successfully evolve walking robot controllers [2].

There are three components to a MathEngine powered simulation - the dynamics simulator, the collision detector, and the visual renderer. The dynamics simulator allows the user to create particles with mass in 3D space, and apply forces to them. It uses a first order integrator to generate physically realistic motions of

the particles. Particles can be connected via joints to create articulated bodies. The joints, which themselves occupy a point in 3D space, impose constraints on the degrees of freedom which the particles they connect can move through. The types of joints available include ball and socket, hinge, and prismatic. The dynamics simulator calculates the next position of the particles when it is called with a given step size. The step size is the period of time which the dynamics engine is to simulate. A large step size produces unrealistic behaviour (e.g. it is not possible to perform collision detection until the step has finished). Too small a step size is computationally intensive and will also produce unrealistic behaviour since the changes in particle positions will be small or non existent, which nullifies the first order integrator. In these experiments a step size of 5ms was used, which produces a dynamics simulation running at 200Hz. When the simulation is visualised the scene is rendered once for every 4 dynamic steps, which creates a 50Hz (i.e. same as PAL television) animation.

The collision detector allows the programmer to associate geometric primitives (cube, sphere etc.) with particles. After the dynamics step the new positions of the particles are known. The geometry of these particles can now be checked for intersections, which indicate that a collision has taken place. To prevent the bodies from passing through each other a contact constraint then needs to be created, which applies an instant restitution velocity to both particles so that after the next dynamics step their geometric primitives will be touching, but not intersecting. The MathEngine Collision Toolkit allows the calling program to create materials with different properties (such as coefficients of friction and restitution) and to declare that a geometric body is fashioned from this material. When a contact occurs the material properties of the two colliding bodies are used to calculate appropriate friction and restitution forces.

The MathEngine viewer uses the particle positions calculated by the dynamics engine and the geometry information required by the collision detection engine to visualise the scene. It contains the necessary procedures to display and render the geometric primitives in their correct position using either OpenGL or Windows Direct 3D. The user can change the camera position and orientation in real-time using the mouse. The simulator provides a mode in which it accesses each rendered frame using an X-window call and saves it into a sequence of files which can later be encoded into an MPEG movie using the Berkeley 'mpeg_encode' software. By accessing the output X-Windows frame directly hardware accelerated 3D can be used, and the frames can be grabbed as they are viewed in real time.

The simulated robot has motors on its hip and knee joints that enable it to walk. These motors are connected to outputs from the neural network simulator. The robot is capable of perceiving a small amount of information about its environment. It has proprioception (information about the bodies internal state) from sensing the angles of its limbs, and exteroception (information about the world) from its spherical feet which can sense when they touch the ground.

Two different models for the hip and knee joint motors were developed. The

first is the built-in MathEngine motor model. The desired velocity of the motor is controlled with a signal in the range 0 (reverse), 0.5 (off), and 1 (forward), i.e. the desired velocity is scaled from [0,1] to [-VMax,+Vmax]. The actual velocity of the changing joint angle is constrained by the attached physical bodies. Since the total force which the motors are capable of generating is limited a quick change in desired velocity produces a slower change in the limb speed.

To enable an easier evolutionary path the signal for the hip motors is interpreted so a value of 0 pushes the leg down, and a value of 1 raises it. When an outgoing neuron is unconnected its activation will be 0. This makes the default behaviour of the network be to push the hips down and raise itself upright. The evolutionary process has a head start because it doesn't have to discover how to stand - any evolved circuit will have to work with the initial standing behaviour rather than against a fallen behavior.

The second motor model is a proportional derivative controller. Again the signal is in the range [0,1], which is the output range of the simulated neurons. This time the signal is scaled to the range $[0,\frac{\pi}{4}]$ and is used as the desired limb angle. The actual torque force exerted on the limb joint is calculated as: $T = K_p(\theta_d - \theta) + K_d\frac{d(\theta_d-\theta)}{dt}$where K_p is the proportional constant, θ_d is the desired angle, θ is the current angle, and k_d is the derivative constant. The proportional and derivative constants were experimented with by trial and error until some realistic values were found. The way this was implemented in MathEngine was by manipulating MathEngine's internal motor model, which produced very unrealistic behaviour for some of these values. The model still displays problems with oscillating limbs, caused by inertia of the error derivative (and presumably inertia from the physical bodies attached to the joint), and drooping of the hip joint due to the inability of the motor to maintain a constant force to act against gravity.

3.4 Neural Simulator

The neural network simulator provides all the functionality needed to simulate different types of networks. It is called by the robot simulator which provides it with a copy of the input sensor array from the robot. These inputs are mapped onto the outputs of 'IncomingNeuron' objects. The activity of all the neurons is then calculated, causing the sensor signals to propagate through the network. The outputs of all neurons are updated with the newly calculated values and the output values from 'OutgoingNeuron' objects (or other neurons in the networks output list) are returned to the simulator. The simulator provides a few different neuron models - incoming and outgoing as already mentioned, as well as sigmoidal and continuous time first order.

The SNNS (Stuttgart Neural Network Simulator) library was originally used to simulate neural networks in the application. This was a recommended library that appeared to have a large amount of documentation and supporting programs. Unfortunately there were a few problems with its use in this project.

Most of the documentation related to the use of the supporting programs, and there was very little information about the simulator 'kernel' that did the actual network processing. The API wasn't sufficiently documented and I found myself having to look at the source code of the other applications to figure out how to use the library. The supporting programs weren't as useful as I had originally thought they might be; I found that I couldn't easily visualise the networks, and the graphical tools were unwieldy. A much greater problem was that the library and tools seemed to be designed for working with sets of training data and using algorithms like back-propagation to train networks rather than evolve them. The simulator itself wouldn't simulate networks with certain characteristics, such as unconnected inputs, which created problems because not all of the 18 inputs provided by the robot simulator would be connected. The simulator only supported sigmoid neurons, and was written in C and hence not object oriented. There was no obvious way to extend the simulator with different neuron models without changing it. In light of all these problems I decided to write my own neural network simulator in C++, with an object oriented design that allowed easy extension, interfacing, and sub-classing of useful base classes (like 'Neuron').

This effort resulted in a new library that is more robust, allows networks with different types of neurons, and can easily be extended. All of the public API is documented in the programmers user manual. It has been designed to be as simple as possible to use. A short example follows (// denotes a C++ comment):

```
SigmoidNeuron a=new SigmoidNeuron(1.5); // create neuron a
SigmoidNeuron b=new SigmoidNeuron(-5);  // create neuron b
Network net;                              // create network
net.add(a);                             // add a to network
net.add(b);                             // add b to network
Connect con(a,b,10.0);       // create connect from a to b
net.add(con);                 // add connection to network
net.save("sample.net");          // save network to file
```

Neural networks can be loaded and saved to files. The file format is quite simple, as so:

```
name: walk
num_inputs: 18
num_outputs: 13
num_neurons: 92
num_connects: 146
-- mapping of output array to neurons
output_map: 79,80,81,82,83,84,85,86,87,88,89,90,91
-- number type par1,par2,par3..
-- (0-17 are inputs)
```

```
neurons:
18 sigmoid 5.000000
19 sigmoid 5.000000
20 sigmoid 5.000000
-- target source1:weight1,source2:weight2..
connections:
18 6:-10.000000
19 7:-10.000000
20 8:-10.000000
```

An ASCII text format was chosen as this provides an easy way to analyse the networks. Simply dumping the network from memory to a binary file would have resulted in a format that was unreadable to humans. The file format support comments as lines that begin with '–'. A 3 letter extension '.net' is used to identify these files.

Before the simulation begins the output of each neuron is randomly reset. This ensures that the starting state of the network is unknown. We wish to evolve networks whose dynamics cause them to act as stable attractors to the cyclic states. When the network is in a random state the neuron biases and connection weights should ensure that it will fall quickly towards the nearest state that is part of its pattern generating cycle. Acting as a dynamically stable attractor helps the network to cope with irregularities in signals, and with large scale trauma such as over excitation of neurons, which can produce epileptic style over-activity in the network, and lack of activity due to damage or unconsciousness in the network.

3.5 Genetic Algorithm

Three different classes of genetic algorithm were implemented:

1. Evolution of totally interconnected neurons using a simple copy and mutate strategy for reproduction.
2. The method used in [11] to evolve SGOCE programs whilst encouraging ecological niches.
3. A co-evolution strategy that attempts to evolve two separate populations of genes and chromosomes at the same time.

The first networks to be evolved were totally interconnected networks of neurons. The number of processing neurons was arbitrarily fixed at 13. This was an educated estimate of the number of neurons needed - other experiments have shown that the number of neurons required for an interconnected network can be as low as 10 (Reil & Husbands biped), or as high as 100 (Richard Reeves

quadruped). Two different types of population were evolved - one where the networks consist of sigmoidal neurons, and the other with networks consisting of continuous time 1st order neurons.

A population of 50 chromosomes was evolved for 200 generations. The initial population is created with randomised neuron biases and adaption rates, and randomised connection weights. Limits are set for each parameter - bias is in [-4,4], adaption rate (for 1st ordre neurons) [0.5,5], and connections [16,16], as suggested in [2].

After each set of fitness evaluations the next generation is created by rank based selection with parents being in the top 10%. Identical copies of the top 10% replace the lowest 10% of the population, but only if they have a greater fitness. The whole population (including the new children) except for the top 10% is then mutated by replacing the parameter values (bias, adapt rate, connection weight) with random values drawn from the appropriate ranges. The mutation of each parameter is done with probability of 0.001, i.e. on average 1 parameter will be replaced for every 1000 parameters. The chromosome consists of 13 neuron biases, 13 adaption rates, and 390 connection weights. This means the chromosome consists of 416 32-bit floating point values, and is 13312 bits long. Many chromosomes will not have any of their parameter values replaced and will continue on into the next generation unaltered.

Preserving the top 10% allows good genetic material to remain in the population, whilst the other 90% is used to search through other possible solutions. Only replacing solutions that have a worse fitness value ensures that the population can spread out and thoroughly search plateaus in the fitness surface. The average chromosome is not likely to live for more than 2 generations without being altered or replaced.

Totally connected networks of sigmoid and continuous time neurons were evolved for motor models 1 and 2. As model 2 uses proportional derivative actuators evolution was expected to be quicker because timing is not as important - the network only has to output the next angle for each walking state. With the model 1 motor the network has to switch between analogue values of 0 for reverse, 0.5 for stop, and 1 for forward. To produce walking behaviour the network has to time these carefully, and coordinate changes between different limbs. The second model allows the body to correct itself automatically when a limb is out of position (e.g. if the robot stumbles). This negates the need for detection and correction of involuntary limb movements by the neural network, thus making its task easier.

The evolution of SGOCE programs was done using a strategy presented by Kodjabachian and Meyer in [11]. Sensory neurons and motoneurones are placed on the 2D developmental substrate. The positions of these neurons is chosen so that there is a direct correlation between the morphology of the robot and its limbs, and the relative positions of the neurons on the substrate. A program is evolved to run on 6 precursor cells which are placed on the substrate. The cells are placed so that there are 3 on each side of the line of symmetry which runs

down the centre of the robot. The cells in each row of 3 are an equal distance apart. Figure 6 shows the developmental substrate used. The evolved program is interpreted in the context of each precursor cell. The cells have an orientation which differs depending on which side of the line of symmetry they are on. The purpose of this is so that the angles of the 'GROW' and 'DRAW' commands will be opposite for cells on different sides of the substrate, so the developed networks will be symmetrical (e.g. figure 8).

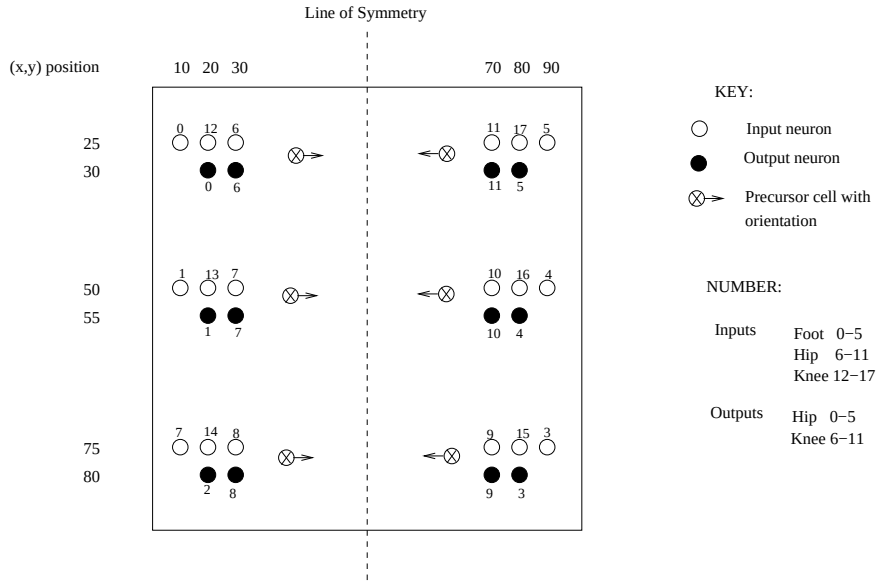


Figure 6: SGOCE developmental substrate

The evolutionary algorithm used by the SGOCE is quite different from the traditional one. The population is viewed as a circle of individuals. To create a child a point is first chosen in the circle, and then a 2-tournament selection scheme is used to select a parent from the local neighbourhood around the point. The program is then reproduced using three randomly selected genetic operators of recombination, mutation, and pertubation as described previously to create the child. A two-tournament selection scheme is again used in the neighbourhood, this time to find a weak individual. This weak individual is replaced by the child, which is inserted into the same position in the population.

The use of tournament selection and localised neighbourhoods is intended to allow the evolution of ecological niches in the population. Genes will spread throughout the population more slowly, and most of the population will be isolated, at least for a while, from the effects of super-fit individuals. This system is analogous to biological evolution, in which physical distances mean creatures are more likely to mate with a partner who lives locally than one far away. In nature this does indeed produce localised evolution, where evolution produces

individuals better adapted to their immediate environment. Eventually this process can even produce a split in the species, making the two groups no longer able to mate.

The neighbourhood evolutionary algorithm as described cannot be used in parallel. The child of each pair must be inserted into the population before another pair can be selected for mating. This is unfortunate as it makes evolution a sequential process that is not as able to take advantage of distributed computation. The evolver does attempt to optimise this by evaluating individuals in parallel, but the constant introduction of new children means the evolver is considerably slower than other algorithms.

The third algorithm to be implemented was co-evolution of a SGOCE program genepool and the chromosomes, which are simply sets of these genes. The gene cells are now no longer fixed to the 6 precursor cell positions on the substrate. Instead each gene consists of a program, a position on the substrate, and an orientation. Each generation consists of evolving the two populations seperately. First all of the chromosomes have their fitness evaluated. This fitness can be used directly to create the next generation of chromosomes by reproduction, where the lowest fitness chromosomes are replaced with copies of the top 10%, and mutation, where genes are randomly added, deleted and replaced. The next population of genes is created by reproducing the SGOCE programs in the normal way, and mutating the starting position. Equal credit assignment is used to distribute the evaluated chromosome fitnesses to their component genes. For a chromosome with 2 genes each one will receive half of the total chromosome fitness. A genes fitness is therefore dependant on how many chromosomes use it, how many other genes these chromosomes also use, and how well the chromosome performs.

Crossover and mutation is used to create the next generation of chromosomes. In each generation the bottom 20% are overwritten by children of the top 10%. Two parents are chosen randomly from the top 10%, and crossover is used to produce a child containing genes from both parents. There is now a 10% chance of each mutation operator being used. There are three operators - add a gene, which increases the length of the chromosome, remove a gene, which randomly removes one of the genes present, and replace a gene, which randomly selects a gene and replaces it with one chosen randomly from the genepool. The child chromosome is now used to replace some chromosome in the bottom 20% of the population.

Mutation is used to create the next generation of genes. The weakest 20% of the population is replaced with children from the top 20%. The normal SGOCE operators are used - recombine, which adds program code from a second parent, mutate1 which replaces a randomly selected subtree of the program with a randomly generated one, and mutate2 which pertubates a certain number (drawn from B(n,p)) of program parameters. The operation of these functions is exactly as described previously. The only addition to the SGOCE operators is a mutation function which alters the starting location of the precursor cell

on the developmental substrate. This is done with probability 0.5. The x and y values are modified with a Gaussian distribution centred around the current value, with standard deviation 5, and then clipped to the substrate boundaries.

The main difference between this co-evolution scheme and the real world is that in the real world there is no central repository of genes. Individual copies of genes are carried by each individual. This is inefficient, but it is the only way autonomous creatures could exist. It will produce a wider search of the gene space since each creature has their own copies that are subject to mutation during reproduction. When better genes are found it can take a long time for them to become distributed throughout the population. Hopefully using a centralised database of genes will allow rapid updating of the whole population as better genes are discovered. It will also be more space efficient.

Making genes part of a population subjects them to the same evolutionary pressures as all of the other parameters. Different genes will compete against each other for survival. They will interfere with each others operation to the extent of negatively impacting each other, but on the other hand they won't damage the fitness of the overall chromosome or else they will be hurting themselves. It is possible that sets of genes will evolve to produce cooperative behaviour, much as happens in the real world. This is a significant advancement over the general SGOCE algorithm which just evolves single programs with a limited grammar. These programs are subject to the same grammar, but since there are many of them operating in parallel it doesn't pose as much of a restriction to the complexity of the developed networks.

Each GA uses the same fitness function. The fitness value is the distance walked by the robot along the x-axis during the simulation. The robot begins facing along the x-axis so it will naturally walk along it when it goes forward. Although there is no explicit penalty for touching the ground with its torso, losing balance and turning all of these will be penalised implicitly as the robot won't be able to walk as far. It is more desirable to use a simple fitness function like this rather than decomposing the fitness into seperate components as this assumes no knowledge of how an evolved controller will work. Decomposing the fitness into concepts such as leg movement, cyclic networks, etc. forces networks to go through stages of evolution which satisfy these constraints. On the other hand, there has to be an evolutionary path from our starting random chromosomes to ones which are capable of completing the given tasks. Walking is a simple enough task that it shouldn't be necessary for staged fitness to be used.

3.6 Distributed Processing

Carrying out physically accurate simulations with highly constrained bodies was computationally intensive. Each simulated second took around 1.75 real seconds to calculate (on a PIII-700 PC). This meant it would take 105 real seconds to simulate 60 seconds. A long simulation time was desirable as it gives a more accurate fitness value. If a shorter time is used the robot may

perform well but then stop or fall over. A single genetic algorithm run can require hundreds of individuals to be evaluated for many generations, which will take a long time. For example, a run with 100 chromosomes per population evolved for 200 generations with 60 simulated second evaluations would require 24.3 days of CPU time! This was unacceptably high, so I decided to write a distributed processing module that would run fitness evaluation simulations on unused computers connected to the network [1].

Writing low level networking routines to carry out the distributed processing was considered but the time for application development was limited. I decided to use a publically available library, of which two seemed to be prominent - MPI, the "Message Passing Interface", and PVM, the "Parallel Virtual Machine". PVM seemed to have a simpler interface, and was supplied with a terminal program that would make debugging the programs easier as remote hosts and processes could be listed and added to. PVM is a C library, and is not object oriented. Several C++ wrapper classes were available but they seemed to be overly complex and not capable of doing the kind of distribution of client tasks the genetic algorithm needed. With this in mind, I decided to write my own object oriented wrapper which resulted in the 'Pvm' class.

A master-slave network model is used. The evolution engine is linked to the Pvm object so that it takes the master role. It performs all distributed processing related tasks; it maintains active host and task lists, and spawns unassigned tasks on hosts with free processor time. If there is no local 'pvmd' daemon running it will start one and then start remote 'pvmd' servers by using the 'ssh' command to execute them remotely on hosts in its hostfile. If hosts crash or become disconnected from the network their running tasks are reassigned. The client is an independent fitness testing program that runs the robot simulation using the supplied parameters. The parameters allow the calling server task to dictate which neural network to use, the length of the simulation, etc. Once it has performed the simulation the client returns the fitness value as a floating point number. The server can then dispatch a new task for that host.

The Pvm module has been shown to scale well; it has been successfully used for evolution runs on a network of over 90 hosts.

3.7 Python Script Interpreter

As development of the application progressed it became apparent that a lot of time was being wasted repeating the modify-compile-deploy-test cycle. There were a lot of parameters to supply to the executable program via the command line (number of chromosomes, generations, hosts file etc.). Changing some options, such as the type of evolution to use, required modifying the source code and recompiling. Both the client and server executables then had to be transferred to the PVM network machines before they could be tested. Replacing

[1] the Linux machine halls at dcs.ed.ac.uk

the front-end of the application with an script language interpreter seemed like a good way to eliminate the need for command line arguments, and at the same time provide a more versatile interface that would skip the modify, recompile and redistribute cycle.

There are many scripting languages that would be suitable candidates for this application. Python was chosen as it is object oriented, and fits in better with the notion of C++ classes than other languages. Like Tcl, it has an interactive interface that the user can type commands and queries into. By this stage of the development all of the C++ classes were complete. Interfacing them to an interpreter for another language could have been quite a struggle. Luckily, an application known as SWIG [13], the "Simplified Wrapper and Interface Generator", has been written for exactly this purpose. As Python is a relatively clean language there is no explicit concept of pointers and memory allocation (in a similar way to Java). This introduced a few problems, as did some unusual data types passed by the C++ class methods. After working through these a Python interpreter linked to all the C++ classes was created.

The interpreted Python script replaces the functionality of the C++ main() function. It must do everything that the main() function would normally do when it interprets the command line parameters. This means setting up and configuring the evolver, population, Pvm or simulator, and fitness tester classes. A typical script looks like (# is the comment symbol in Python):

```
from evolve import *              # use my C++ libraries
from time import *                # use time module
srand(time())                     # init random generator
e=CoEvolver()                     # create population
e.createInitialPopulation(5)     # create 5 chromosomes
s=Simulator()                     # create simulator
f=FitnessTest()                   # create fitness tester
f.setUnitEvaluate(s)              # tell fitness tester to
                                  # use the simulator
f.setEvalTime(30)                 # use 30 second evaluations
e.setFittest(f)                   # tell evolver to use
                                  # fitness tester
e.evolve(100)                     # evolve 100 generations
```

The syntax is very similar to C++, allowing the wrapped C++ libraries to be used with the same class and method names. There is no impact on the performance of the application using scripts instead of compiled code. Only a very small amount of the CPU time is spent in the main() function, since the majority of the time is spent in the simulator called by the evolver.

Every C++ class is wrapped by Python shadow classes. These shadow classes are full Python classes - they can be inherited as a base class from other Python classes, allowing the user to extend any class with Python implementation, and

they fully support method overriding for virtualised classes. This allows easy extension of classes like 'Evolver', 'Neuron', and others in the future.

3.8 Other Useful Programs Developed

A 'viewer' program was developed to visualise the robot simulation using the neural network file supplied as a command line argument. It uses the MathEngine renderer which can be manipulated with the mouse to control viewing distance and orientation. The user can manually pan the camera to keep up with the walking robot, or an automatic robot tracking mode can be enabled.

Two programs were developed to visualise the evolved networks. They both make use of other software to plot the network and save it as an encapsulated Postscript graphic.

net2ps_dot converts files in the neural network '.net' format into Postscript using AT&T's 'dot' graphing software [14]. This application can display the topology of networks. There is no geometry with the associated networks, so the output networks tend to have the neurons arranged so as to minimise connection distances. Figure 7 shows the manually developed walking controller discussed later. Inputs are arranged at the top of the image, and outputs at the bottom. Excitory connections are shown as full lines and inhibitory connections are dotted lines. You should be able to spot some similarities between this and the hand-crafted schematic in figure 9 as they are the same network!

net2ps_plot converts files in the neural network '.net' format into Postscript using the GNU Plotutils library [15]. This application displays the geometry of evolved networks, so its only useful for SGOCE type networks where neurons each have a unique position on a 2D plane. Figure 8 shows an example from an evolved symmetrical network. Clear boxes depict 'IncomingNeurons', and filled are 'OutgoingNeurons'. The placement of these boxes corresponds roughly to the robot morphology - the inner boxes correspond to the hip sensors and motors, and the middle ones to the knees, and the outer ones to the feet.

3.9 Summary

This chapter has covered the design and implementation of the application. A rigid object-oriented design methodology was followed, along with strictly defined public class interfaces and methods. This helped in creating a functional system that was free of any obvious bugs. To ensure correctness of the code the system now had to be thoroughly tested.

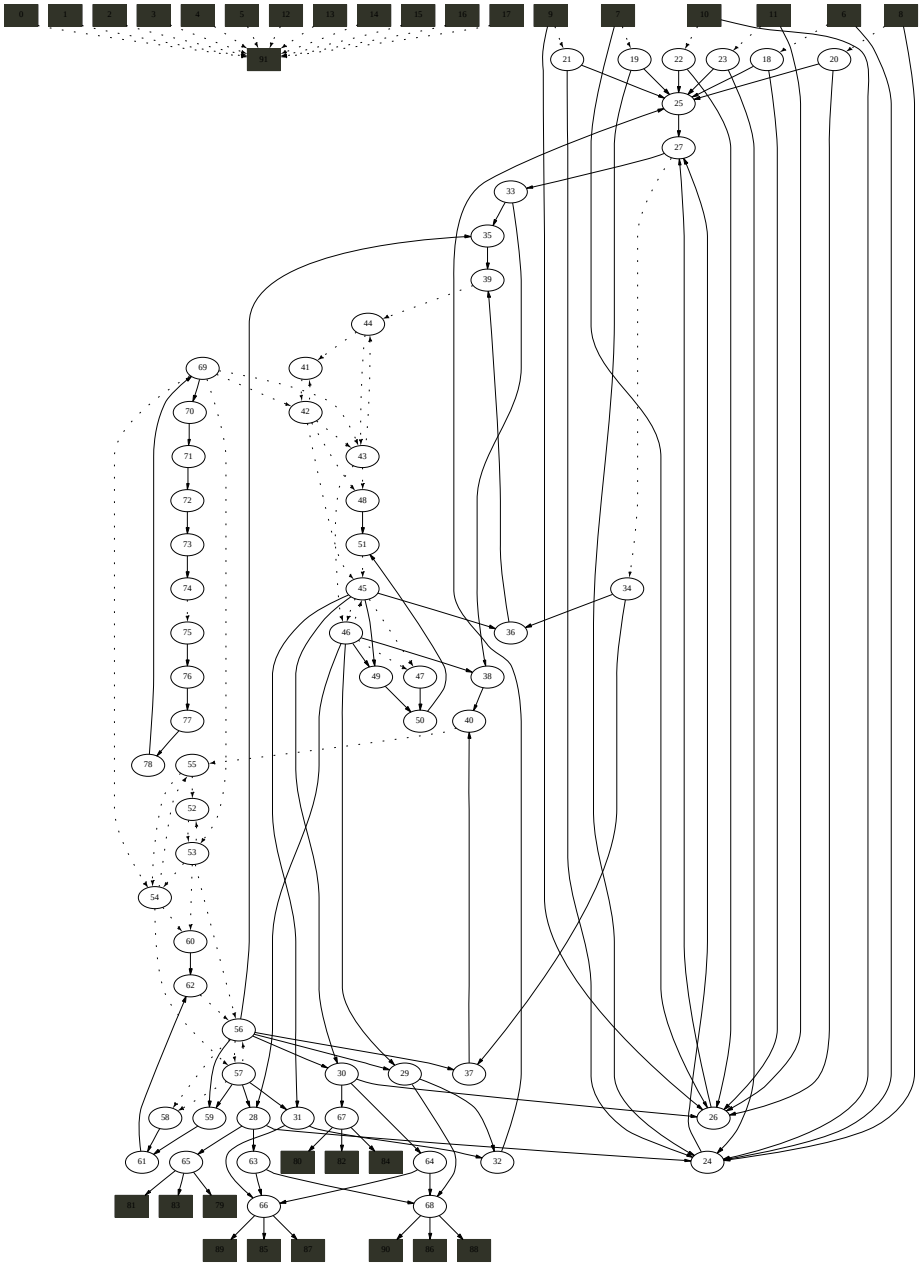


Figure 7: Manual controller network visualised using 'net2ps_dot'

4 Testing

It was desirable that the system was bug-free and functioning correctly. During development the basic functionality of individual components were tested using short functions before they were integrated with the larger program code. Once the components were developed the system was tested as a whole by designing a large neural network capable of controlling walking.

4.1 Correctness Testing

The system was tested thoroughly whilst under development, and a controller was manually designed to perform end-to-end testing on the simulator and the network. The neural network simulator was tested by designing simple, small networks and using a calculator to work out their outputs for some given inputs. The networks were then transferred into the format of the simulator, and simulated with the given inputs. The results were compared with the precalculated values to ensure they were the same.

Testing of the neural and physical simulators together was carried out by creating a digital circuit that could be simulated with a neural network. This was initially done to test the SNNS neural simulator with my physics code. After some debugging it worked correctly. When SNNS was replaced with my own neural simulator testing was done to ensure that it generated identical outputs. The new neural simulator was then tested along with the physics simulation to ensure that they worked correctly together.

Debugging versions of the MathEngine libraries are provided which output information from the physics solver. The robot simulation was initially written

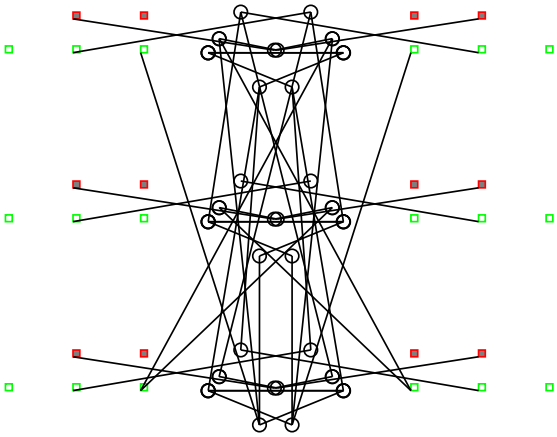


Figure 8: Example of visualising network geometry with 'net2ps_plot'.

as several large functions, and was not object oriented. When the MathEngine debugging libraries were linked instead of the usual ones they revealed several errors, the source of which was not apparent (e.g. the solver was complaining that the collision detector was creating contacts with non-unit vectors!). This was unexpected, as with the non-debug libraries the simulation appeared to be fine. Redesigning and rewriting the code to be object oriented, with thorough checking via the debug libraries throughout the rewrite, solved these problems.

The evolutionary routines have been tested during end-to-end testing of the system as a whole. Due to the random nature of evolutionary runs it is difficult to perform any unit testing. However, the code is simpler than the neural and physical simulations, and therefore less prone to error. The SGOCE coding was tested by creating small programs whose operation was predictable using manual means. The programs were then converted into networks and visualised. Correctness was ensured as the networks clearly corresponded to their predicted appearance. The visualisations showed connection weights and neuron biases, and there was clear inheritance of values from the parent.

The PVM network code was tested by running evaluation tasks on remote hosts and comparing the returned fitness value with the expected fitness value. They would not be identical due to the random initial state of the network, but they should be similar. It was observed that two parallel evaluations of the same network would return the same fitness value, which was unexpected as the randomness should have made them slightly different. This was eventually found to be caused by the random seed being the value returned by time() - the number of seconds since January 1st 1970. Since the evaluation programs were called at the same time, and the clocks on the network computers were synchronised, they both set the random seed to the same value. This was solved by XORing the number of seconds with a value read from /dev/random, which is unique for each Linux system.

4.2 Manual design

A neural network was designed to produce walking in the robot simulation. As there are no design methodologies for manually creating a neural network controller the network was built from logic gates which are simulated using neurons. A state machine is used to control the outputs which drive the motoneurones. There are 18 inputs in the simulator (12 limb angle sensors and 6 foot contact sensors). This circuit uses the hip angle sensors to drive transitions between the states.

The robot produces tripod walking, with three feet in contact with the ground at once. There are four states - lifting 3 legs up, swinging these legs forwards whilst driving the other 3 backward, and these two states repeated with the role of the groups of legs reversed.

The circuit uses a Moore machine design, where the current outputs and next

state are determined by the current state. There are some problems with simulating a circuit in this way. The neural network simulator simulates propagation delays in the neurons. This produces a behaviour where connected components don't necessarily fall into a stable state as connected silicon would, which can lead to oscillations in the circuit. There are also problems with the flip-flops not storing a stable value. To solve this two approaches were taken. The first was to initialise all of the neuron outputs to 0 before the simulator is first run. This ensures that the starting state is stable and predictable. Extra gates and connections were added to the flip-flops to eliminate internal oscillations, and the setup and hold times (in neuron propagation delays) of the flip-flops were calculated. A clock signal was then added which respected these constraints. The clock consists of a series of buffers and a single inverter. The buffers define the clock period while the inverter ensures that the signal will change after this period.

4.3 Circuit Schematic

The numbers above each neuron refer to its number in the file description. Standard logic gate symbols are used here, but these are in fact logic gates simulated by neurons as shown in the following section. The input signals are H0 upto H5, which are hip angle sensors. The outputs are the H024, K024, H135 and K135 signals, which go to the hip and knee motors. Each signal goes to three motors, which corresponds to the tripod walking this robot displays. The S0 and S1 signals which feedback from the right hand side of the circuit to the flip-flops carry the next state. This is clocked into the flip-flops when the CLK signal goes high. It takes some time for the new state to propagate through the circuit before the motor outputs change. The four AND gates labeled S0,S1,S2,S3 signal which walking state we are currently in. There are four corresponding states which each drive the legs in a different way. One of these gates will be active to indicate that we are in that state, the other three will be outputting 0. The next state is calculated by the circuitry preceding the S0 and S1 signals at the top right. The 6 input AND gate (number 28) outputs a CS signal, which goes high when the conditions are met for the circuit to change state. Note the use of a buffer (34) in parallel with the NOT gate here (35) - this ensures the 2 bits S0 and S1 which indicate the next state switch at the same time, so the state clocked in for the next cycle is always either the current one or the next one.

The inputs and outputs are arranged into arrays as follows:

INPUTS

0-5	foot contact
6-11	hip angle
12-18	knee angle

OUTPUTS

0-5	hip motor
6-11	knee motor
12	dummy neuron (not connected)

The reason for the dummy neuron is that the neural network simulator I originally used (SNNS) didn't allow inputs to exist unless they were connected to outputs. Rather than change the robot simulator I opted to add this extra output and connect all of the unused inputs (foot contacts and knee angles) to it. It isn't connected to any part of the robot simulator; its signal is discarded.

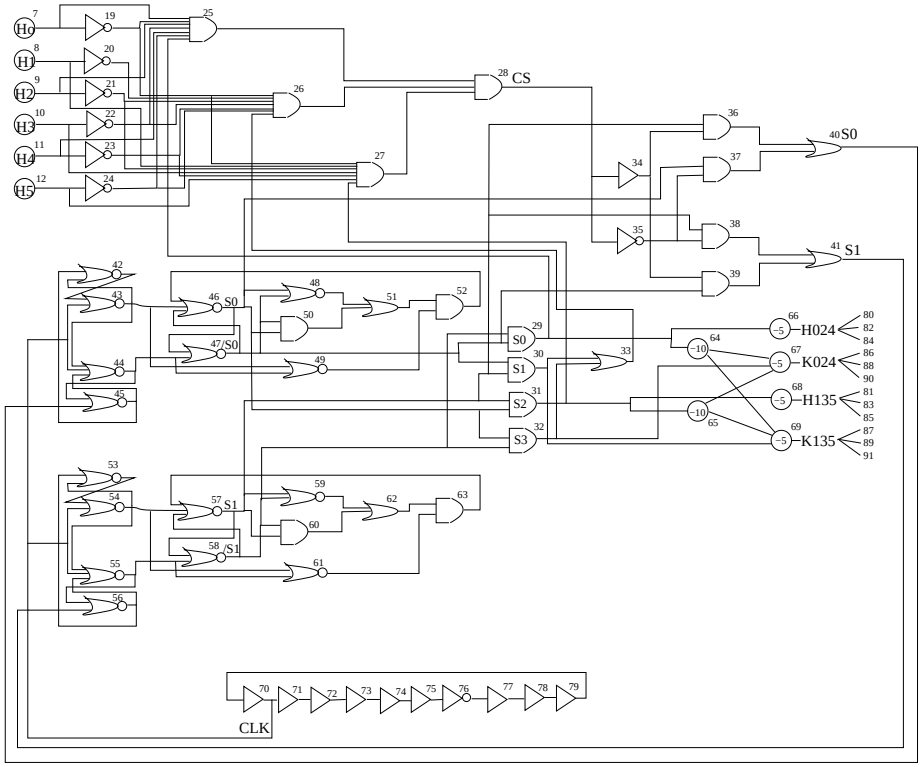
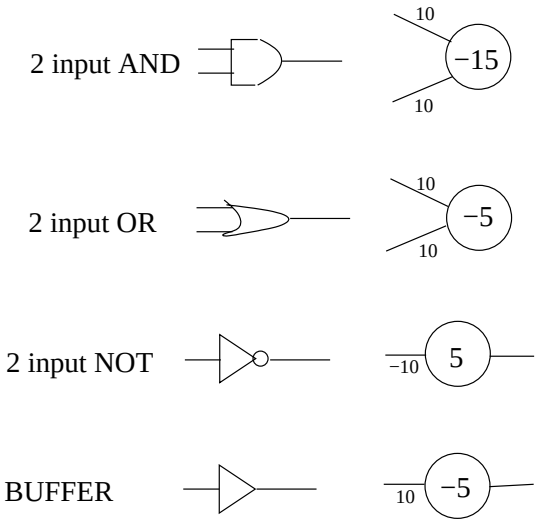


Figure 9: Manual Controller Schematic

4.4 Circuit Components

Each logic gate is simulated using a neuron with certain weights on its inputs.



4.5 Performance

The manual controller performs well, but not optimally. Over a 20 second fitness evaluation it can walk 11.75 meters. Its tripod gait is effective; it uses all legs, and doesn't lose its balance. Its speed is 2.115Km/hour. Due to its digital logic design this network will have problems that an evolved network won't. There is no redundancy - removing a single neuron will cause the circuit to fail. Walking is slow as the circuit waits for the robots legs to reach their most extreme angles before switching state. This can be seen in the way that the robot lifts its legs up to a 45◦ angle before moving them forward when simply clearing the ground would suffice.

5 Experimentation

A set of experiments, consisting of evolutionary runs with different parameters, was devised. It was intended that these experiments would compare the different approaches, contrasting the two motor models, different neuron models, different chromosome encodings, and the different genetic algorithms. These experiments form the basis of conclusions that are later drawn.

5.1 Experimental Setup

A single evolutionary run was carried out for each set of parameters. It may have been beneficial to carry out more runs and average the results, however, this should not be necessary in a good GA - the parallel aspects of the search

should ensure that it always arrives at an optimum result, avoiding problems like premature convergence and unfocused search that may hamper an inferior GA.

The scripting language was used to make a single script for each run. Most of the evolutionary runs were carried out using distributed evaluations on a network of over 90 hosts. The exception to this was the SGOCE implementation. As described in [11] the sequential evolution was carried out on single host. This required a considerably longer running time than the other algorithms.

Each genetic algorithm was run for 200 generations with 50 individuals. The robot simulation was carried out for 20 seconds, which is enough time to ensure that walking is cyclic and will repeat forever rather than coming to a halt. The sequential evolutions of the SGOCE programs were carried out for 15,000 generations. In each of these generations only 1 child is changed, unlike the other algorithms which change many children. This made the increased number of generations necessary.

The different parameter settings were:

1. Totally interconnected network with sigmoidal neurons and motor model 1
2. Totally interconnected network with sigmoidal neurons and motor model 2
3. Totally interconnected network with continuous time neurons and motor model 1
4. Totally interconnected network with continuous time neurons and motor model 2
5. SGOCE network with motor model 1
6. SGOCE network with motor model 2
7. Co-Evolved chromosomes with SGOCE genes with motor model 2

5.2 Results

These figures show the progress of fitness over time for the experiments. The upper curve in each imagine shows the maximum fitness of that generation, the middle curve shows the mean fitness, and the lower curve shows the minimum fitness.

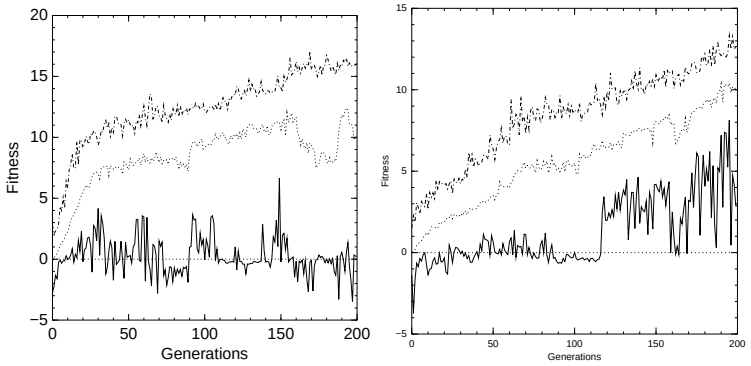


Figure 10: Evolving interconnected sigmoid networks with motor 1 (left) and 2 (right)

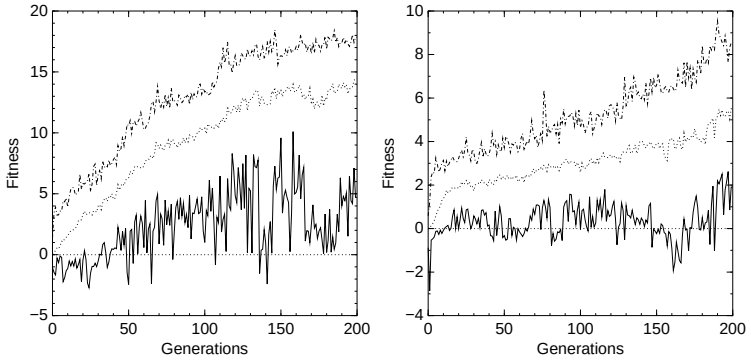


Figure 11: Evolving interconnected continuous time networks with motor 1 (left) and 2 (right)

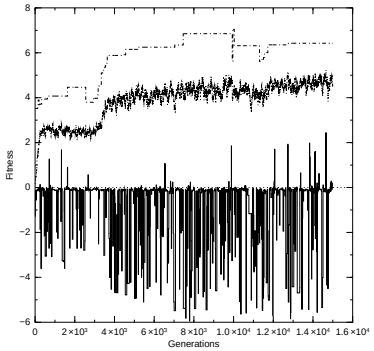


Figure 12: Evolving SGOCE programs with motor model 2

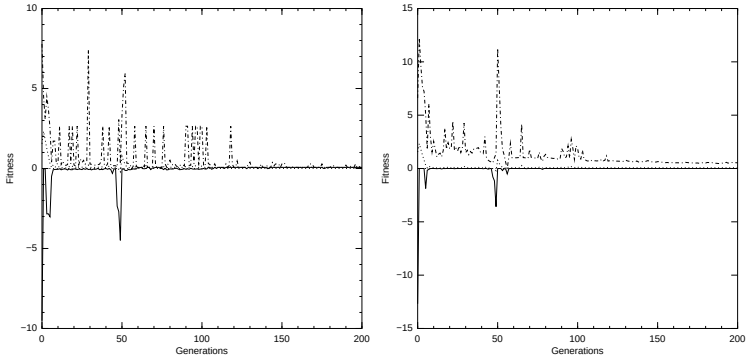


Figure 13: Co-Evolution of chromosomes (left) and SGOCE genes (right) with motor 2

Table 1: Distance travelled by the best individual in the final population

Experiment	Distance (metres)
1: interconnected sigmoid motor 1	15.985
2 : interconnected sigmoid motor 2	12.658
3 : interconnected ct motor 1	17.857
4 : interconnected ct motor 2	8.526
5 : SGOCE motor 1	—
6 : SGOCE motor 2	6.424
7 : Co-Evolution	0.07

5.3 Discussion

Experiment 5 was aborted prematurely after it became apparent that no evolution was taking place. Due to the large time requirements of this algorithm it was decided not to continue as after several thousand generations none of the robot simulations had walked any distance. Interestingly, the GA had no problems evolving controllers for the same experimental setup with the 2nd motor model. I believe there are two reasons why the GA was unable to evolve any solution for the 1st motor model. As you will recall, the 1st model requires signals of 0 for reverse, 0.5 for stop, and 1 for forward. The default output of a continuous time neuron, as used in the SGOCE networks, is 0.5 for no incoming connections. This means the default state is to not move any legs. Furthermore, the fitness function rewards actual travel across the plane, and not a combination of leg motion and travel as used in [11]. This severely impacts the GA, leaving it with a highly constrained evolutionary path. It must discover how to connect to the sensory neurons, how to process the signals, and to connect to the outputs with some reasonable weighted connections. It is too much to expect this to happen through random search alone. The GA requires a clearer evolutionary path, which can be given explicity as a multi-parameter fitness function, or through staged evolution. In this respect the deficiencies of the 2nd motor model may actually be working to its advantage. The limbs tend to oscillate around the angles output by the neural network. This is jump-starting the evolution process by providing it with limbs that already move in rhythmic motions.

The neural networks are always initialised into a random state before being simulated. The fact that these networks are able to walk successfully shows that they are acting as dynamically stable attractors. No matter what the starting state is, they will always fall into a stable attractor state that is part of the walking cycle. This is a significant property of biological neural networks, and it is good that the evolved networks are capable of the same behaviour.

The biggest disappointment was in the performance of the co-evolution GA. Only motor model 2 was used due to the previous failure of model 1 to produce any walking with SGOCE programs. There are some clear spikes in the two graphs, with the most successful individuals reaching a distance of over 10 metres, which is far better than the maximum of 6.424 achieved using the SGOCE encoding alone. It is unclear why the population is peaks and then falls so dramatically. One hypothesis is that the nature of the credit assignment works to level the playing field between good and bad genes rather than promoting good ones.

A well performing gene will be used in an increasing number of chromosomes. Unfortunately, many of these new chromosomes which inherit the good gene will perform badly, which will decrease the assigned fitness of the gene. Several chromosomes with a strong negative fitness could even cause the gene to itself have a negative fitness. There is a small amount of evidence to support this hypothesis;

upon examining the evolved population of chromosomes many of contained very similar sets of genes. They also seemed to contain more genes than would be expected given the chromosome fitness. Chromosomes with smaller numbers of genes will assign a greater proportion of their fitness to each one. Thus it can be expected that highly performing chromosomes will consist of fewer, higher fitness genes. However, chromosomes in the final population tended to contain at least 5 genes. I think that this is a result of the use of crossover as the method of reproduction.

As the chromosomes have an arbitrary length the crossover point is chosen to be somewhere in the body of the minimum sized parent. This means children will be the same size as either of the parents, which would not account for chromosome growth. However, crossover has a tendancy to copy the junk genes that are between pairs of good genes. This is directly analogous to human DNA, where it is believed that around 93% of the DNA is unnecessary. The bias of the credit assignment towards shorter length chromosomes doesn't seem to be enough to stop this effect.

The use of sigmoid neurons versus continuous time neurons doesn't seem to have had a great impact. Sigmoid neuron networks performed worse with motor 1, but better with motor 2. It seems that normal walking behaviour doesn't require a complex neuron model.

It is interesting to note that for totally connected networks the fitness over time tends to increase for the maximum, mean, and minimum fitness chromosomes. Contrast this with the curves for the SGOCE evolution. That the maximum curve is much straighter is due to the way in which the GA works - a networks performance isn't reevaluated unless it has been altered in some way. The mean curve increases and decreases locally, but on the whole shows a gradual increase. The surprise is in the minimum curve. Unlike the other algorithms it shows no curve at all. There is a massive variation in the minimum performance, sometimes jumping from 0 metres (no movement) to past -5 metres (reverse movement). This is probably due to the developmental coding. If a new child performs particularly badly it will cause a quick drop in the curve. The child will be short lived due to its low fitness, and when it is replaced there will be a quick jump.

The SGOCE encoding makes the generation of particularly bad children more likely than with totally connected networks. A bad neuron or a bad connection weight will be compensated for by the dynamics of the rest of the network. A bad SGOCE parameter will often have a more disastrous effect, producing entire sets of neurons that are displaced on the substrate, or other similar wide ranging effects.

The 1st motor model performed better than the 2nd in the totally connected networks. As already discussed, it didn't perform as well with the SGOCE encoding. I believe that this is due to the first model being more accurate and providing the neural network with a greater degree of control over the body's actions. A perfectly implemented proportional derivative actuator should work

as well as the first motor model. However, the horizontal oscillations around the output knee angle and the drooping of the legs as the hips reached the desired vertical angle both indicate problems with the motor model that are due to incorrect constant parameters and the way in which the actuator was implemented inside MathEngine (as a hack of the internal MathEngine motor model). The other alternative hypothesis is that it wasn't neccessary to have a control process between the motoneurone and the motor. The neural network may be capable of carrying out this function by itself. Unfortunately there is no evidence to support this conclusion.

Figure 14 shows the top fitness network in the final SGOCE population. Clear squares indicate sensory neurons, and filled ones motoneurones. As can clearly be seen the network is symmetrical about the vertical line of symmetry. The majority of the connections are horizontal, which indicates that they are performing some processing of the input signals with the central neurons to generate the output signals. There are a small number of vertical connections which may perform synchronisation between the pairs of legs. For the exact meaning of the sensory and motoneurone layout consult the substrate layout in figure 6.

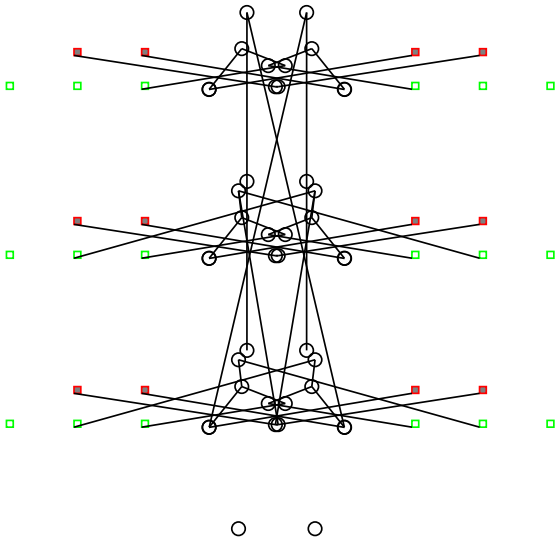


Figure 14: Top fitness network in the final SGOCE population

6 Conclusions

This project set out to produce evolved neural network controllers for a simulated robot. This has been successfully achieved, producing a variety of neural networks which utilise different neuron models and different types of connectedness. Different evolution strategies were employed, including an advanced

developmental encoding that allows complex networks to be created from sequences of simple instructions.

The SGOCE encoding proved to be useful for developing sets of neurons with limited amounts of interconnections. In many ways it is more similar to the coding of DNA than direct encodings are. Having a totally interconnected network on the scale used here (13 neurons) is not a problem - a typical biological neuron is connected to thousands of others. However, as evolved networks scale to become larger a system like this will have to be employed to prevent calculating the next state of the network from becoming an intractable problem. It is likely that, as in biological networks, neurons that are close in the physical space will also have a similar function in the network. Thus a developmental encoding that biases physically close neurons to be connected is more likely to be useful in the real world than one that allows neurons to be equally connected regardless of their location.

The biggest problem facing evolutionary techniques is the computational power required to simulate the real world. Even if the physics simulations were perfect, allowing us to recreate the world exactly (as in the film 'The Matrix'), we would still need to be able to accelerate the simulations by many orders of magnitude to be able to evolve human level intelligence in a reasonable amount of time. We can never hope to attain the level of processing power that would be required for this kind of simulation. We need to be able to successfully translate evolved controllers into real world robots so that they can achieve some useful task. For this to be successful the tasks that we want them to achieve, and the problems that they may encounter on the way, have to be accurately simulated. Given that we will never have the computational power for fast, accurate simulation of complex environments a limit is placed on the kind of tasks that we will be able to evolve controllers for. The ideal solution is to evolve controllers which are evaluated in the real world. Unfortunately this presents even more problems! We would need a mechanical body for each individual, or some way to evaluate them using a shared body. Evaluations will be done in real time, making the evaluation of complex tasks by many agents a time consuming process, and unlike biological evolution we don't have the luxury of 3.5 billion years of evaluation time.

Artificial evolution alone will not be enough to create complex controllers that display intelligent behaviours. Its usefulness will be in combination with other processes that are well understood to create systems that perform very specific tasks. If biological brains can be divided into sub-modules, which it appears they are, and the functions are well understood then it will be possible to create genetic algorithms that evolve networks that carry out the same functional operations.

6.1 Future Work

There are many ways in which this project could be extended. The simulated environment could be made more complex, with a 3D terrain, obstacles, and other agents to interact with. Access to a large amount of sensory information is essential for a complex life form to function. Vision, audio, and smell could be integrated into the simulation and provided to the robot through arrays of sensory neurons.

A 3D version of the SGOCE encoding could be used alongside 3D sensory neurons. This would allow more accurate simulation of senses such as touch that rely on the geometry of the robots body. Sensory input would be already mapped and organised into two dimensions like skin is on animals, which makes evolving networks that interface to it much easier. More advanced neural network models could then be used, such as a 3D version of the 'GasNet' networks. Evolution of the robots morphology is an obvious next step, with the precursor cells placed within different body parts, so there is no arbitrary separation of body and controller in the creatures genome.

The evolutionary approaches tried here can be applied to other robots with more complex controller requirements. In particular, it would be interesting to see how the various encodings and neural architectures compare when used to create controllers for a walking biped. Staged evolution could be used with the aim of producing evolved higher level controllers. Walking forwards is the first step (no pun intended) towards controllers that are capable of more advanced behaviours. Using staged evolution provides an easy way to build higher level controllers that are capable of tasks like walking and turning on demand that can support even higher level behaviours like walking to a given location and tracking sources of sensory stimulus.

The SGOCE encoding isn't complex enough to support the kind of development we see from human DNA. Stem cells present in the foetus can reproduce, and turn into any other cell in the body depending on their current location. This type of encoding would be a lot more powerful as it allows arbitrary depths of cell resolution within a space. Limits on the resolution would have to be defined, e.g. 1000 neurons/cm^3. Support for some depth of recursion in the instruction set would be useful. The encoding used in Sim's creatures uses recursion with a specified maximum depth to prevent infinite or impractically large recursion.

As mentioned previously, biological evolution is incredibly powerful as it evolves not only the chromosome genotype, but also the way in which the genes are decoded into functions that build the body and controller. This could be emulated in artificial evolution to gain the advantages of not having a prespecified coding. Hopefully this would allow evolution to discover a coding and corresponding phenotype mapping that are better than the human designed systems. It may be the case that this kind of evolution would allow the encoding and building of the creature to approach the complexity and elegance of natural DNA. The evolved chromosome would need to contain some kind of program that specifies

a method of reproduction. Simply copying the parent chromosome should be a method that will be quickly found by random search, and of course once it is found individuals that contain it will quickly come to dominate the population. The encoding should allow different methods for reproduction, such as crossover, and multi-parent recombination to be represented.

One way to imagine this coding is to think of the chromosome as being the tape of a Turing machine. There is no arbitrary separation of code and data; this is expected to happen to some extent through evolution to prevent the reproduction code from being disrupted as other parts of the genome are mutated. The start of the reproduction program within the chromosome may have to be indicated with a special symbol. Once the program is encountered it will be interpreted according to set rules. This may seem like a cop-out since these rules aren't evolved, but it isn't really since the interpretation of the program is more analogous to the interpretation of a natural chromosome according to the physical rules of our world. A simple grammar ought to suffice, allowing commands like 'loop' for iteration, 'prob' for randomised execution, 'right' to move the read head, and 'copy' to concatenate so many bits onto the child chromosome. Implementing a simple copy and mutate strategy is surprisingly easy:

```
loop
  left
loop
  prob 0.01
    copy 0
  prob 0.01
    copy 1
  copy
```

This program simply states that first we should loop left infinitely. This will be terminated when we hit the left side of our chromosome. We then enter the second loop. With a low probability we set the next bit in the child to a 0 or 1, otherwise we copy the existing bit. It is easy to imagine that a small program like this will be discovered randomly by evolution. Subject to evolution itself, it is furthermore easy to see that this program could eventually evolve into a multi-parent reproduction process with optimal mutation rates. Initially it may appear that a mutation rate of 0 would be best, however this is not the case as the children of chromosomes with 0 mutation will be beaten by children of chromosomes who are mutating, and hence improving, their fitness. The instructions in the chromosome can be extended to add commands which grow the morphology and controller of the robot. These commands would be executed by the interpreter when it is moving through the chromosome. Evolution of the interpreted program would hopefully allow the discovery of more powerful representations such as run-length encoding. Allowing multiple development threads to be evolved inside the chromosome will be more robust than single

threads, and more analogous to biology. To my knowledge a system of evolving reproduction programs and developmental codings as part of the chromosome hasn't been tried before.

All of the software developed for this project is written in standard C++, using only the standard libraries or publically available open source libraries. This makes porting and extending the project very easy. Unfortunately, the MathEngine library is proprietary, and only available for x86 based PCs and videogame systems. MathEngine is also no longer available for academic use. A new library is being commercialised by a spin-off company from MathEngine Plc., but it is no longer free for academic use. With this in mind one future path would be to switch the physics simulator to a free, open source project like Dynamo [16].

Dynamo was originally considered for this project but rejected as there was a lack of demonstration code. Since this time others have started to develop open source software (primarily games) that use the Dynamo engine, making example code available. The dynamo library will compile on practically any system. Its physics is expected to be at least as good as MathEngine, and it has the advantage that the programmer can select to use a 1st order integrator, as MathEngine uses, or a more accurate 3rd order integrator. Dynamo is also compatible with an open source collision detection library. An object oriented abstraction class for physic and collision detection needs to be created. Once this is done implementations of the class could be written using any physics simulator, and it would be totally transparent to the end-user. Is it probable that evolutions carried out with a mixed variety of simulators will produce controllers capable of acting more robustly in the real world. They are also less likely to exploit inaccuracies in the physics of any particular simulation.

The distributed networking code has been designed to be robust, but could be further improved. The evaluation program will return a 'host unsuitable' message if it is getting less than a certain amount of processor time. It would be better if hosts were dynamically queried and then ranked according to the processing power that they have free. Tasks could then monitor the amount of processing time that they are receiving from the local host, and migrate to another host if there would be a performance gain.

The fitness function could be altered to take into account different things that we wish to optimise inside the network. We may wish to evolve networks which minimise the number of neurons, or the distances that signals have to travel between neurons. Similarly, we may wish to minimise the power consumption of the network, something that is important in biological lifeforms which may have to survive through extended periods without food. This could be done by calculating the power consumption of the network while it is running, and then using it as part of the fitness function.

Artificial life is the simulation of complete environments, with competing and cooperating agents, as opposed to this project which uses genetic algorithms with simple fitness functions. Evolution in artificial life is much more complex,

but also more interesting. Nobody has yet simulated a distributed artificial life ecology that evolves controllers and morphology. One way of gaining access to the vast amounts of computing power needed is to release the software as a screen-saver, similar to the "SETI" (Search for Extra Terrestrial Intelligence [33]) software. When a client PC isn't in use the screen-saver becomes active and starts communicating with other clients via the Internet. Distributed evolution like this has been done in the 'Tierra' [32] project, but that seeks only to evolve binary code and not complete creatures. Combining a system like this with a 3D evolution environment, like Sims work but with creatures competing for resources, would be very interesting and would provide a visually attractive screen-saver at the same time! It is likely that given enough sensory data and evolution time these creatures would begin to display traits that we associate with real lifeforms, such as cooperative behaviour, communication, and learning.

The neural network simulator developed for this project only includes simulations of two neuron models, and there is no support for the more complex modulation and neurotransmitter mechanisms that are present in the brain. This style of neural network is common in artificial intelligence research, but neuroscientists tend to use much more detailed and biologically accurate models. It would useful if this project could be interfaced to a more complete neural simulator which focuses on accurate biological models, such as the Genesis simulator developed at Caltech [23].

Recent developments in stem cell research have shown that the growth of neural cells in an embryo is stimulated by the self organisation of the network, i.e. cell growth is a property of the brain learning, and receiving sensory data. It would be interesting to try an approach like this with a genetic algorithm. A developmental encoding would be used, but either the instructions or the way in which they are interpreted would be affected by environmental stimulus. The robot would need to go through a period of growth, in which it learns and adapts to its body, before it could be evaluated. Since all bodies differ in their exact mechanics giving the network the ability to adapt dynamically to its sensory input and motoneurone outputs would allow more adaptable and robust controllers to be evolved.

How big would a chromosome need to grow before it could approach human levels of intelligence? As one observer put it:

> Of the 750 megabytes of human DNA, the vast majority is believed to be junk and 98% is identical to chimpanzee DNA, with perhaps 1% being concerned with intelligence - leaving 7.5 megabytes to specify, not the actual wiring of the brain, but the neuroanatomy of areas and maps and pathways, and the initial tiling patterns and learning algorithms for neurons and minicolumns and macrocolumns. [34]

Even though the largest programs evolved so far are only a few kilobytes long, it is quite amazing to think that a mere 7.5 megabytes of instruction (albeit in a

highly compact form) could account for human intelligence. Evolving programs of this size is certainly within our grasp, or will be with the coming advances in computational power, and will help to bring the dream of true AI one step closer.

References

[1] Baerlocher, P.,
http://ligwww.epfl.ch/baerloch.html

[2] Reil, T., Husbands, P. (2000)
Evolution of Central Pattern Generators for Bipedal Walking in a Real-Time physics Environment
http://users.ox.ac.uk/~quee0818/files/reil_husbands2000.zip

[3] Reeve, R. (1999)
Generating walking behaviours in legged robots
http://www.dai.ed.ac.uk/~richardr

[4] Goldberg, D.E. (1989)
Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, MA, USA.

[5] Husbands, P., Smith, T., Jakobi, N., O'Shea, M. (1998)
Better Living Through Chemistry: Evolving GasNets for Robot Control
http://www.cogs.susx.ac.uk/users/toms/GasNets/CONNSCI98/

[6] Husbands, P., Smith, T., Jakobi, N., O'Shea, M., Anderson, J., Philippides, A. (1998)
Brains, Gases and Robots
http://www.cogs.susx.ac.uk/users/toms/GasNets/ICANN98A/

[7] Ijspeert, A.J.,
http://rana.usc.edu:8376/~ijspeert

[8] NewScientist (2000)
Half Fish, Half Robot
http://www.newscientist.com/news/news_224233.html

[9] Beer, R.D., Gallagher, J.C. (1999)
Evolution and Analysis of Dynamical Neural Networks for Agents Integrating Vision, Locomotion, and Short-Term memory
http://vorlon.cwru.edu/~beer/Papers/GECCO99.ps.gz

[10] Biological neurons (diagram borrowed from here)
http://vv.carleton.ca/~neil/neural/neuron-a.html

[11] Kodjabachian, J., Meyer, J.A.,
Evolution and Development of Neural Controllers for Locomotion, Gradient-Avoidance, and Obstacle-Avoidance in Artificial Insects
http://www.biologie.ens.fr/fr/animatlab/perso/meyer/kodja/loco2D.ps.gz

[12] Sims, K. (1994)
Evolving 3D Morphology and Behaviour by Competition. Artificial Life IV Proceedings, MIT Press, pp.28-39.
http://www.genarts.com/karl/papers/alife94.pdf

[13] Beazley, R.,
Simplified Wrapper and Interface Generator
http://www.swig.org

[14] AT&T
'dot' is part of the Graphviz package.
http://www.research.att.com/sw/tools/graphviz/

[15] GNU Plotutils
http://www.gnu.org/software/plotutils/

[16] Dynamic Motion Library (Dynamo) physics simulation software
http://www.win.tue.nl/~bartb/dynamo/

[17] The Honda Humanoid Walking Robot
http://world.honda.com/robot/

[18] Studies on Leg/Foot functions of the Honda robot
http://world.honda.com/robot/technology/

[19] Hornby, G.S., Takamura, S., Yokono, J., Hanagata, O., Yamamoto, T., Fujita, M.,
Evolving Robust Gaits with AIBO
http://www.demo.cs.brandeis.edu/papers/hornby_icra00.ps.gz

[20] Sony AIBO quadruped entertainment robot
http://www.aibo.com

[21] Sony SDR-3X walking biped entertainment robot
http://www.sony.co.jp/en/SonyInfo/News/Press/200011/00-057E2/index.html

[22] Auke, A.J., Kodjabachian, J.,
Evolution and Development of a Central Pattern Generator for the Swimming of a Lamprey
http://rio.lip6.fr/papers/Ijspeert_Kod_AL99.ps.gz

[23] GENESIS - GEneral NEural SImulation System.
http://www.bbb.caltech.edu/GENESIS/

[24] Grillner, S., Wallen, P., (1985)
Central pattern generators for locomotion, with special reference to vertebrates. Annual review Neuroscience 8: 233-61.

[25] Burgess, R.J.,
Human Locomotion
http://www.vector-mobilization.com/evolution/human_gait.html

[26] Mitchell, T.M.,
Machine Learning section 4.1.1 (McGraw-Hill)

[27] New Scientist
Is this the mother of all brain cells?
http://www.newscientist.com/nsplus/insight/clone/stem/
isthisthemother.html

[28] Braitenberg, V.,
Vehicles: Experiments in Synthetic Psychology

[29] Ross, P., (1998)
Genetic Algorithms and Genetic Programming. Dept. AI, Edinburgh University.

[30] Forrest, S., Mitchell, M., (1992)
Relative Building Block Fitness and the Building Block Hypothesis
http://www.santafe.edu/~mm/Forrest-Mitchell-FOGA.ps

[31] Plaisted, D.,
How old is Humanity?
http://www.creationinthecrossfire.com/documents/
HowOldHumanity/HowOldIsHumunity.html

[32] Ray, T.,
Tierra Digital Evolution
http://www.isd.atr.co.jp/~ray/tierra/

[33] Search for Extra-Terrestrial Life (SETI)
http://www.seti.org

[34] The Singularity Institute
An Introduction to the Singularity
http://www.singinst.org/printable-intro.html

[35] Heesch, D.,
Doxygen documentation system
http://www.stack.nl/~dimitri/doxygen/