Minimal Simulations For Evolutionary Robotics

Nick Jakobi

Submitted for the degree of D. Phil.

University of Sussex

May, 1998

Declaration

I hereby declare that this thesis has not been submitted, either in the same or different form, to this or any other university for a degree.

Signature:

Acknowledgements

My thanks go first and foremost to Phil Husbands, my supervisor, for his guidance, his patience, his ability to see that there might be something of worth in even the most confused ramblings, and his ‘just another quick one then’. Thanks to Inman Harvey, Mike Wheeler, Peter deBoucier, Joseph Faith, Giles Mayley, Seth Bullock and all the other members of Alergic and the COGS formation drinking team for crucial discussion, inspiration and theoretical debate. Thanks also to Sally and Stephen Jakobi for financial and moral support, John Byrd and John Anderson for running in the Downs, Suzy Levy for stress-relief, Damon Shaw on saxophone, T.C. for Sunday night pints, Derek Parkinson for showing me how to groan properly and efficiently, Adam Bockrath for importing mathematical rigour and Peter Stuer for saying it couldn’t be done.

Minimal Simulations For Evolutionary Robotics

Nick Jakobi

Summary

For several years now, various researchers have endeavoured to apply artificial evolution to the automatic design of control systems for robots. One of the major challenges they face is how the fitness of evolving controllers should be tested when each evolutionary run typically involves hundreds of thousands of such assessments. This thesis puts forward new techniques for evolving control systems for real robots using easy-to-build, fast-running simulations.

It begins with a tutorial in state-of-the-art Evolutionary Robotics and discusses the best types of neural network, encoding scheme, genetic algorithm and genetic operators to use - and how to use them. Several novel types are introduced, and their relative merits over previous approaches are discussed.

After analysing the conditions that must be met by controllers if they are to perform the same behaviour in reality as they do in simulation, a new methodology is proposed for building minimal simulations within which controllers that meet these conditions will successfully transfer into reality. Techniques are then put forward for forcing controllers that evolve to be reliably fit within such minimal simulations to meet these conditions.

Four sets of experiments are reported, all involving minimal simulations. Controllers were evolved for a small mobile robot that could solve a T-maze in response to a light cue, target recognition and approach behaviours were evolved for a visually guided mobile robot, walking and obstacle-avoiding behaviours were evolved for an eight-legged robot and motion-tracking behaviours were evolved for a simple panning camera head. In all four cases, the evolution of complex robot behaviours that would have taken many months to evolve if fitness evaluations had been performed in reality was performed in a matter of hours, and controllers that evolved to be reliably fit in simulation displayed extremely robust behaviour when downloaded into reality.

Submitted for the degree of D. Phil.

University of Sussex

May, 1998

Contents

Chapter 1

Introduction

Controlling robots is tricky. It is relatively cheap and easy to hook a powerful computer up to sensor and motor systems these days, but programming it to perform more than a fraction of the behaviours displayed by simple animals is presently beyond the state of the art. Over the past few years, a growing number of researchers have become disillusioned with traditional engineering approaches to this problem (Beer 1990; Brooks 1991b; Brooks 1991c) and have started looking to biology for their inspiration. In particular, they have been studying the processes that give rise to the behaviours of animals acting in the world, and asking whether abstracted versions of these processes might be made to give rise to similar behaviours in robots (Cliff 1991; Brooks 1991a).

One such process is evolution, and the discipline that seeks to apply abstracted evolutionary processes to the design of controllers for robots is known as Evolutionary Robotics. This term was perhaps first used by Husbands and Harvey (1992), but the idea of using artificial evolution to automatically design controllers for robots can be found in several papers from around the same time (Barhen, Dress, and Jorgensen 1987; Viola 1988; Garis 1991; Brooks 1992; Harvey and Husbands 1992). Early work in the field was concerned entirely with evolving controllers for simulated robots (Beer and Gallagher 1992; Cliff, Husbands, and Harvey 1993b; Higuchi, Niwa, Tanaka, Iba, Garis, and Furuya 1992). Since the first few practical examples (Harvey, Husbands, and Cliff 1994; Floreano and Mondada 1994; Yamanuchi and Beer 1994), however, there have been an increasing number of papers dealing explicitly with the issues involved in evolving controllers for *real* robots (Nolfi, Floreano, Miglino, and Mondada 1994; Jakobi, Husbands, and Harvey 1995; Miglino, Lund, and Nolfi 1995; Jakobi 1997; Jakobi 1998), and that is what this thesis is about.

There are many different ways in which controllers can and have been evolved for robots. In order to give the reader an intuitive idea of what such a process might look like, a simple example of how a particular robot behaviour may be automatically generated using artificial evolution shall now be described. A population of perhaps 100 initially random controllers are evaluated according to a test function based on the controller's ability to perform the desired behaviour, and a score is attributed to each. The controllers with the highest scores are then subjected to various reproductive processes which create a brand new population of 'offspring' controllers that are similar to their 'parents' but which differ in small random ways. This new population is then used to make yet another population in the same way, and so on round in a cycle, generation after

generation. The fact that the controllers of each new generation are less than perfect copies of the best controllers from the generation before means that most of them will not score as highly on the test function. However, the slight random variations also mean that some may score more on the test function, and these will then be selected to act as parents for the next generation. In this way, controllers can evolve to achieve higher and higher scores on the test function through the dual actions of random variation and selection. The idea is that eventually, after many generations, controllers evolve that can perform the desired behaviour. Magic!

But of course it is not magic and as several authors have pointed out (Brooks 1992; Harvey and Husbands 1992; Mataric and Cliff 1996; Nolfi, Floreano, Miglino, and Mondada 1994), there are many big questions that need answers if Evolutionary Robotics is to progress beyond the proof of concept stage. One of the most urgent of these (in that if it is not answered, Evolutionary Robotics is not going to progress very far at all) concerns how evolving controllers should best be evaluated. If they are tested using real robots in the real world, then this has to be done in real time, and the evolution of complex behaviours will take a prohibitively long time. If controllers are tested using simulations then the amount of modelling necessary to ensure that evolved controllers work on the real robot may mean that the simulation is so complex to design and so computationally expensive that all potential speed advantages over real-world evaluation are lost. How then should controllers be evaluated when testing in both simulation and reality seems fraught with insurmountable problems?

The main contribution of this thesis is to offer an answer to this question. It does this by presenting new ways of thinking about and building simulations for the evaluation of evolving robot controllers. These *minimal simulations* run extremely fast and are trivially easy to build when compared to more conventional types of real-world simulation, yet they are still capable of evolving controllers for real robots. Thus the many advantages of using simulations are preserved while most of the major disadvantages are avoided. I shall now explain the structure of this thesis in detail.

In order to bring the inexpert reader up to speed, the thesis starts with an Evolutionary Robotics tutorial. Aimed at the newcomer to the field, this chapter presents my selection of the best techniques currently available for the artificial evolution of robot controllers, and just as importantly, the circumstances in which they should be applied. Although most of these techniques have been invented by other authors over the past few years (and this chapter serves as a review of these), others are presented here for the first time. It begins with a simple high level example of what an Evolutionary Robotics experiment might look like, and this is used to introduce the topics covered in the rest of the chapter. These include how to decide on what type of controller to use given the nature of the behaviour to be evolved, the best ways of representing these controllers within the evolutionary process, different types of evolutionary algorithms and operators and the circumstances under which they should be used, and how to go about designing a fitness function for evaluating evolving controllers. It is hoped that by the end of this chapter, the reader will be equipped with almost everything they need to know to go about setting up and performing their own Evolutionary Robotics experiments. There is however one thing they will not learn from this chapter: how to apply fitness functions and evaluate evolving controllers in practice. This is the preserve of the next chapter.

Chapter 3 presents the general theory and methodology behind a new way of evaluating evolving robot controllers: minimal simulations. This is done in several stages working from first principles up to a step-by-step guide to building a minimal simulation for the evolution of robot controllers. The chapter starts with an analysis of what it means to say that a controller has transfered from simulation to reality, and this is used to investigate the general conditions under which controllers can and do transfer. The idea of a minimal simulation - the simplest possible type of simulation capable of evolving robot controllers - is then introduced and the general conditions for successful transfer are recast within this context. If this was all there were to minimal simulations, however, then it is unlikely that controllers would evolve to fulfill these conditions. Techniques are therefore proposed for using the evolutionary process itself to force controllers that evolve to perform the desired behaviour within a minimal simulation to also fulfill the conditions for successful transfer into reality. All the various threads are then brought together and summarised to produce a simple step-by-step guide to building a minimal simulation for evolutionary robotics.

Chapter 4 presents a formal treatment of the theory behind minimal simulations. It introduces a logical formalism for reasoning about controllers performing behaviours in environments and *derives* a minimal set of conditions for successfully crossing the reality gap from the same set of assumptions as those made in chapter 3. The fact that these conditions correspond closely to those put forward in chapter 3 provides good evidence for the sound theoretical basis underlying minimal simulations.

Chapters 5, 6, 7 and 8 detail experiments in which controllers were evolved to perform the following robot behaviours:

- **T-maze solving behaviour for a Khepera robot.** A T-maze environment was constructed in which a beam of light could be shone across the the first corridor from either side. Controllers were evolved to guide a Khepera robot through the T-maze, 'remembering' from which side the beam of light was shone and turning down the corresponding corridor arm at the junction.
- **Shape-discrimination behaviour for the gantry robot.** An equilateral triangle and a square of white paper were both stuck onto a long wall of an otherwise black arena. Starting from different positions and orientations, controllers were evolved to steer the gantry robot towards the triangle while ignoring the square.
- **Walking and obstacle-avoiding behaviour for an octopod robot.** Controllers were evolved to make the octopod robot walk around its environment, turning away from objects that fall within range of the IR sensors and backing away from objects that touch the front bumpers and whiskers.
- **Motion-tracking behaviour for a panning camera-head.** Controllers were evolved to make a simple panning camera-head track arbitrarily patterned objects as they moved against arbitrarily patterned backgrounds.

In addition to describing working examples of minimal simulations and how they were designed, these 4 chapters also show how many of the techniques from chapter 2 may be used in practice. In order to bring out the general nature of the underlying methodologies - with respect to both minimal simulations and the rest of the evolutionary machinery - the same explanatory structure (section headings, subsection headings and so on) is used in each chapter.

The thesis concludes in chapter 9 with a few thoughts for the future.

Chapter 2

A Crash Course In Evolutionary Robotics

At present, there are at least as many different methodologies for evolving robot controllers as there are researchers in the field, and probably a whole lot more. This does not matter so much for those whose experience can sort the wheat from the chaff but what of the newcomer to the field? When faced with a mountain of papers, each reporting that the techniques within are of fundamental importance, it is extremely hard for them to know where to start. It might be said that the same is true of any young field, but evolutionary robotics is now mature enough that some techniques are widely used while others have fallen by the way-side. Although it is still too early to say exactly which techniques should be included in a definitive *library* of techniques for evolutionary robotics, it is now possible to say with some objectivity which of the existing techniques should be used in which circumstance.

Although others have written about the main issues of Evolutionary Robotics in a theoretical way (Husbands and Harvey 1992; Brooks 1992; Nolfi, Floreano, Miglino, and Mondada 1994; Kodjabachian and Meyer 1994; Mataric and Cliff 1996), this chapter does something different - it takes the form of a tutorial. Aimed primarily at those in possession of at least a passing acquaintance of the main concepts, such as the majority of researchers from the Artificial Intelligence community at large, this chapter provides both a review of the good work in the field of Evolutionary Robotics and a guide as to how to apply it in practice. In so doing, a reader from the target audience should be left with a good idea of how to go about setting up and performing their own evolutionary robotics experiments.

2.1 An example

Before diving into detailed explanations of the state-of-the-art, this section provides a high-level description of a simple evolutionary robotics example. This will provide intuitive explanations of the main concepts and processes involved and provide a context for the more detailed and low-level discussions later in the chapter. We begin by examining a simple genetic algorithm and show how it can control the processes of selective reproduction and fitness evaluation to evolve controllers for robots. We then go on to deconstruct these latter two processes into their constituent parts, each of which is the subject of a section later in this chapter.

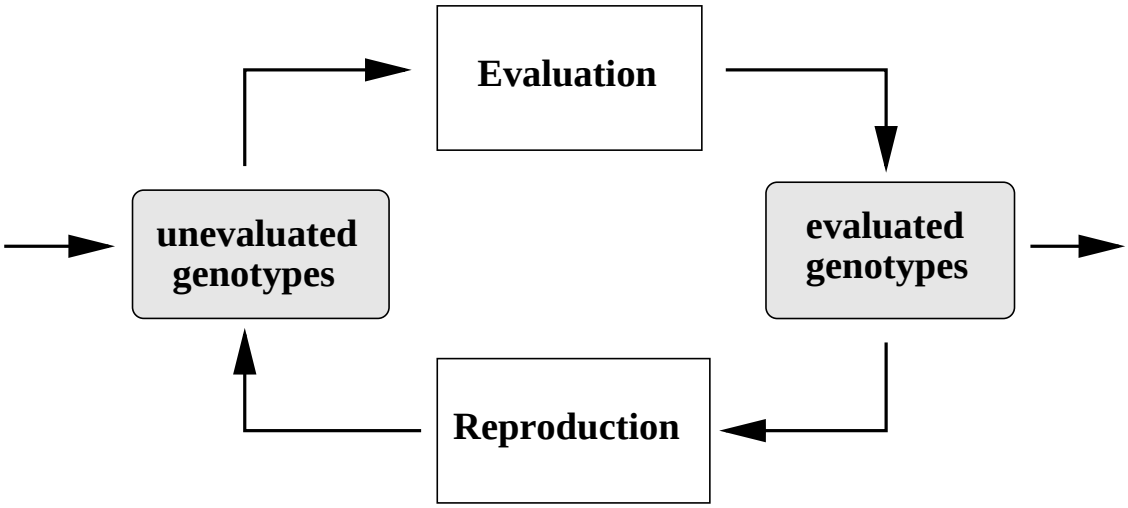

Figure 2.1: *The top level.*

Figure 2.1 shows a block-diagram of the workings of a typical genetic algorithm. This is a generic search technique, invented by Holland (1975), that lies at the heart of most of the present-day evolutionary robotics research. Genetic algorithms do not work directly with robot controllers, however, but instead operate upon populations of 'genotypes', as can be seen from the diagram. For explanatory purposes, genotypes are best thought of as strands of robot controller DNA and they can be represented in a computer program in a number of ways: the simplest of which is probably as binary strings of 1's and 0's. Typically, the initial population of genotypes are randomly generated and unevaluated.

The algorithmic cycle of figure 2.1 starts with a fitness evaluation phase. This results in a fitness value, based on the ability of the robot controller it encodes, being associated with every genotype in the population. When this phase is complete, we are left with a population of evaluated genotypes and the reproduction phase of the algorithm commences. This involves repeatedly selecting probabilistically fit genotypes as parents and applying various genetic operators to them to create offspring genotypes. When enough offspring have been created, we are left with a brand new generation of unevaluated genotypes, and the cycle begins again.

Since the new genotypes of each generation are created from the fittest genotypes of the last generation, the idea is that, over time, the population as a whole becomes fitter and fitter. Eventually, if all goes to plan, genotypes will evolve that code for robot controllers which are able to perform the behaviour(s) we are interested in. We will now look at the processes of evaluation and reproduction in a little more detail.

Evaluation

Figure 2.2 shows a typical ordering of events for the evaluation of a genotype. After it has been selected, a genotype must first be turned into its phenotype ready for testing. The word 'phenotype', in an Artificial Evolution context, refers to the objects that the system has been set up to evolve and hence to those objects whose properties are directly tested during the evaluation phase; in the case of evolutionary robotics, phenotypes are most usually some sort of robot controller.

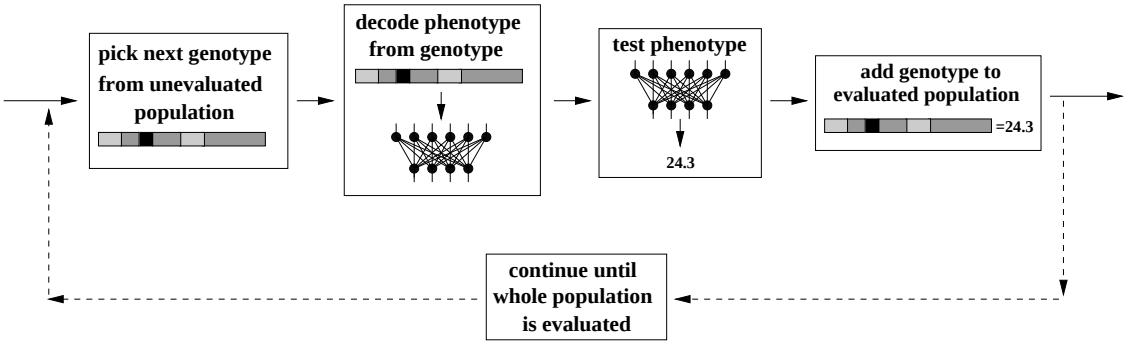

Figure 2.2: *Evaluation.*

The way in which a genotype encodes a phenotype is known as the 'encoding scheme' and may take many different forms. The choice of which scheme to use can have tremendous impact on the success or failure of the evolutionary robotics process as a whole and the pros and cons of a number of different schemes are discussed in section 2.3. A choice also needs to be made as to the type of robot controller we want to evolve. Although evolutionary robotics is not restricted to any one particular type of controller, some are definitely better than others and various different types are discussed in section 2.2.

After a robot controller has been decoded, the next thing to do is to test it. This will normally involve making it control a robot or a simulation of a robot for a period of time and scoring its ability to perform a particular behaviour by way of some automatic fitness function. Issues to do with designing and formulating the fitness function are discussed in section 2.5. This testing is normally the most time consuming part of the entire evolutionary process by several orders of magnitude, and since many hundreds of thousands of tests may need to be performed, exactly how they are done may make the difference between years spent evolving and minutes. This problem is addressed in chapter 3 where a major new methodology is introduced for creating simple fast simulations that can be used to perform these evaluations.

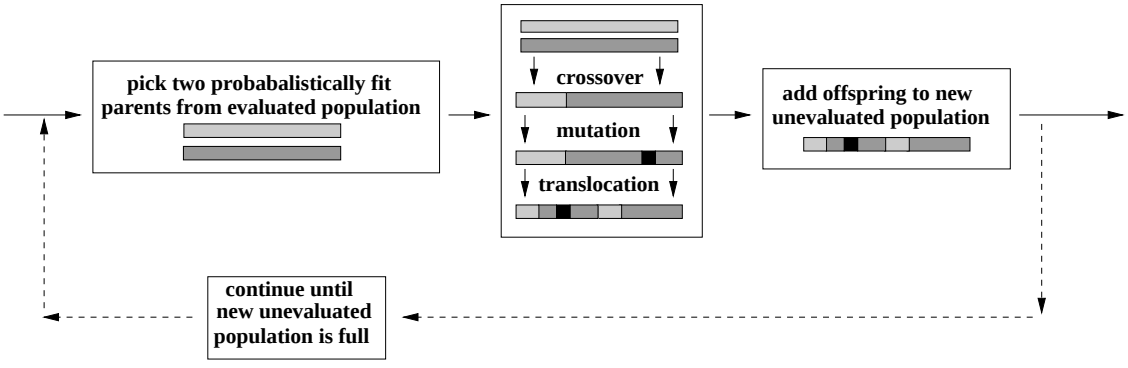

Figure 2.3: *Reproduction.*

Reproduction

Figure 2.3 shows one way in which new genotypes can be created. Two 'parent' genotypes are selected from the current generation on the basis of their high fitness and genetic operators are applied to produce an 'offspring' genotype. In the figure, this procedure occurs in several stages. Firstly, the crossover operator is applied to produce the basic offspring genotype by taking a segment at random from one parent genotype and joining it to a complimentary section taken from the other. A mutation operator is then applied to make small random changes, with low probability, to any and all parts of the genotype. Finally a translocation operator takes a segement at random from the offspring gentoype and moves it to a new random location on the same genotype. The offspring genotype is then added to the next generation and the cycle continues via selection and reproduction until the new generation is complete.

In fact, there are many ways to create offspring genotypes, both in terms of the way in which parents are selected and in terms of the operators applied to them to produce offspring. Section 2.4 looks at different types of genetic algorithms and genetic operators and discusses which are the best suited to the particular problems found in evolutionary robotics.

2.2 What to evolve?
Control architectures for evolutionary robotics

Unlike other machine learning techniques, artificial evolution is not restricted to a particular type of control architecture. In fact, any type of control architecture that can be replicated and mutated is capable of being evolved. The question thus arises, what is the best type of control architecture to use when evolving controllers for robots? This section begins by considering the options available and putting forward arguments for artificial neural networks as the best type of general control architecture. It then goes on to examine which type of artificial neural networks in particular should be used in which circumstance.

Apart from neural networks, various other types of control architecture have been evolved for robots (either simulated or real). The most common are Lisp or Lisp-like programs (Winston 1988). These were first used to control robots by Koza who adapted techniques from his Genetic Programming (Koza 1990; Koza and Rice 1992). These programs are formed from a subset of Lisp or other building blocks which can be represented using the tree structure characteristic of Lisp. It is this tree structure that provides much of the attraction of Genetic Programming techniques since it is particularly robust to the application of genetic operators (Koza 1992). Lisp expressions themselves however display few of the properties listed below that make neural networks attractive for use in evolutionary robotics. Brooks (1992) has proposed keeping the tree representation and using building blocks more suitable for robotics. However, the blocks he suggests are statements in some high-level behaviour language, which seems inappropriate for the type of low-level behaviour evolutionary robotics is currently concerned with. A compromise has been reached by Gruau who has succeeded in representing and evolving artificial neural networks for real robots using Lisp-like tree structures (Gruau 1992; Gruau 1997).

Other types of controller architectures that have been employed include classifier systems (Holland 1987) where genetic algorithms are applied to sets of simple rules for controlling robots (Schults and Grefenstette 1992; Dorigo and Schnepf 1993; Colombetti and Dorigo 1992; Dorigo

and Colombetti 1997). It is hard to argue, however, that a system based on a discontinuous set of discrete rules engaged in the classic match/conflict-resolution/act cycle is the most appropriate type of control architecture for the control of a noisy, real-world, dynamically complex robot.

Although all these alternative architectures have been used with some success on simple robotics problems, there are some good reasons why neural networks in general might be preferable:

- By varying the properties and parameters of the simple processing units used, many different types of functionality can be achieved with the same type of network structure. This means that the same encoding schemes (see section 2.3) can be used independently of the functionality of the control system.
- Using neural networks allows us to implement and test ideas from biology about how neural mechanisms for the generation of behaviour might work. This is a rich source of design inspiration that can be readily coopted for engineering ends as demonstrated by Beer and Gallagher in their design for a hexapod robot controller (Beer and Gallagher 1992). Also, using neural networks allows us to test and refine proposed biological models through the exploration of parameter spaces which may suggest new hypotheses (Ijspeert, Hallam, and Wilshaw 1997).
- There are many other adaptive processes which we may want to use in conjunction with artificial evolution such as various forms of supervised and un-supervised learning. As pointed out by Nolfi, Floreano, Miglino, and Mondada (1994), these are readily implemented within the artificial neural networks paradigm (Floreano and Mondada 1996b; Nolfi, Miglino, and Parisi 1994).
- The behaviours that evolutionary robotics is concerned with at present are low-level behaviours, tightly coupled to the environment through simple, precise feedback loops. Artificial neural networks are ideal for this purpose. The time may come when the behaviours we want to evolve are more complicated and may require higher level primitives, but even if this is the case, there is no reason why these should not be pre-evolved neural assemblies. Various versions of this point can be found elsewhere (Husbands, Harvey, and Cliff 1995; Nolfi, Floreano, Miglino, and Mondada 1994; Cliff, Harvey, and Husbands 1993).

If the arguments above are accepted, then the next question becomes that of which types of neural networks to use for the evolution of which types of behaviour. This is discussed below. In the last part of this section on control architectures for evolutionary robotics we discuss how various aspects of the robot body itself may be evolved in conjunction with the controller to produce complete brain and body systems.

2.2.1 Neural networks for evolutionary robotics

Some adaptive mechanisms for neural networks, such as back-propagation (Hinton, McClelland, and Rumelhart 1986), can only operate when certain classes of neural network are used. Artificial evolution, on the other hand, is not restricted in this way and we are free to dream up and use any type of network we like. Some types of network will evolve better than others, however, and certain types of network are better suited to certain types of behaviour than others. Various different types are discussed below. They are divided up into three behavioural categories, roughly in order of complexity: networks for simple reactive behaviours, networks for simple non-reactive behaviours and networks for dynamically complex non-reactive behaviours.

Networks for reactive behaviours

A reactive behaviour may be defined as one in which the motor output at any particular time is completely determined by the current sensor input. Simple examples include obstacle avoidance and phototaxis[1]. These are amongst the simplest behaviours to evolve since they consist only of a direct mapping from sensor input to motor output. The question of which networks to use for this sort of behaviour, therefore boils down to one of which sort of network is the best at evolving input-output mappings.

The obvious choice of network for input-output mappings is the multi-layered perceptron (Rumelhart, Hinton, and Williams 1986). These have been shown to be capable of producing arbitrary mappings between inputs and outputs and are therefore, in principle, capable of producing *any* reactive behaviour within the mechanical limitations of the particular robotics system. Networks normally consist of three or more layers of units (one input layer, one output layer and a variable number of hidden layers) connected together in a feed forward fashion with activity flowing from inputs to outputs. The input activity A_j of the j_{th} unit is usually calculated from the weighted sum of its inputs

$$A_j = \sum O_i w_{ij} + I_j \tag{2.1}$$

where O_i is the output from the i_{th} neuron, w_{ij} is the weight on the connection from the i_{th} neuron to the j_{th} neuron, and I_j is any external input to the j_{th} neuron from outside the network. The output O_j of the j_{th} neuron is calculated from the input activity A_j according to the sigmoid function

$$O_j = \left(1 - e^{(t_j - A_j)}\right)^{-1} \tag{2.2}$$

where t_j is a threshold constant associated with the j_{th} neuron.

Multi-layered perceptrons have been used by several researchers to evolve simple reactive behaviours such as obstacle avoiding behavior for the Khepera robot (Nolfi, Miglino, and Parisi 1994; Lund 1995). However, multi-layered perceptrons are often capable of much more than we need. They may be capable of arbitrary mappings from inputs to outputs but for most of the reactive behaviours that evolutionary robotics has, up to now, been interested in, the inclusion of a hidden layer will do nothing but vastly increase the space that evolution must search.

There are in fact many different types of networks that serve perfectly well for the vast majority of reactive behaviours. Two-layer perceptrons, for example, use the same sigmoid function of equation 2.2 but have only two layers of units: one for inputs and one for outputs. Not only have these been used by Floreano and Mondada (1994) to perform the same obstacle avoidance behaviour on a Khepera robot as mentioned above, but Lund and Hallam (1996) have shown that they are capable of evolving behaviours such as exploration and homing that, at first sight, one might not even think of as reactive. Other types of networks that have been used to evolve reactive behaviours include biologically inspired continuous-time networks (Hopfield 1984) as used by Beer to evolve simple orienting behaviours on a simulation of a visually guided robot (Beer 1996). These networks are actually of most use in the evolution of non-reactive behaviours, however, and a detailed description of how they work is left until Section 2.2.1 below.

Successful reactive behaviour often necessitates extremely short sensorimotor feedback loops. As long as this feedback is of the correct sign, furthermore, then the exact nature of the function

[1] Although these may also be implemented in non-reactive ways.

that produces it need not be complex. It is for this reason that Braitenberg's vehicles (Braitenberg 1984), which just connect sensors to outputs through a linear gain, are so successful at phototaxis and obstacle avoidance. The same is also true of the type of networks that can be used to evolve the vast majority of simple reactive behaviours. These need not employ sigmoid activation functions or complex time dynamics, simple linear threshold units are usually sufficient, or even binary valued units that can only be 'on' or 'off' (Jakobi 1994; Jakobi 1998).

Networks for simple non-reactive behaviours

A non-reactive behaviour may be defined as one where the motor output at any particular time is not a function of the current input. Thus any behaviour that requires a 'memory' or internal state generated by some past input falls into this category. Also behaviours that do not require any input to generate appropriate output, such as blind walking behaviour in a hexapod robot, are non-reactive. The common defining feature of non-reactive networks is that they have their own internal dynamics which can be influenced by input to the network at any one time but never completely determined by it. Amongst other things, this makes them extremely difficult to analyze once they have evolved although some interesting attempts have been made (Husbands, Harvey, and Cliff 1995; Gallagher and Beer 1993; Beer 1995b; Cliff, Husbands, and Harvey 1993a; Husbands, Harvey, and Cliff 1993a). Networks that are non-reactive will often nevertheless have reactive elements to their behaviour. For example, a network controlling a robot that solves a T-maze in response to a past light-signal it has 'remembered' (see Chapter 5) may still descend the corridor without crashing into the walls in a reactive way. Non-reactive behaviours subsume reactive behaviours, and in general they are more challenging to evolve. The four examples of chapters 5, 6, 7 and 8 are all non-reactive.

In order to have internal dynamics there must be internal mechanisms by which past states of the network can influence present states. The easiest way to achieve this is simply through allowing recurrency in the connectivity of a network. This means that it is possible to find loops of connectivity within the network whereby the input to a neuron depends, at least in part, on some previous output from that same neuron. Thus the state of the network at any particular time will be a function, at least in part, of its state at some previous time.

Various researchers have used recurrency alone (i.e. without altering any of the simple temporal properties of nodes and links) to achieve internal network dynamics sufficient for the evolution of non-reactive behaviour. One elegant example is provided by Floreano and Mondada (1996a) who evolved exploration and homing behaviour for a Khepera robot using a multi-layer perceptron with self-recurrent links on the hidden units. In the experiments the Khepera had a virtual battery which could be charged when it entered the illuminated corner of a rectangular arena. The aim of the behaviour was to explore as much of the arena as possible, follow the light back to the charging area when the battery was close to running out, set out to explore when the battery had charged and so on. Controllers were evolved whose internal dynamics were sufficient to switch between exploration and light-seeking behaviours in such a way that the robot explored the arena efficiently without its batteries ever running out.

Husbands, Harvey and Cliff used recurrent neural networks featuring inherently noisy neuron activation functions to evolve behaviours for a visually guided mobile robot. In (Husbands,

Harvey, Cliff, and Miller 1997) they describe experiments in which controllers evolved that could distinguish a white triangle from a white square and steer the robot towards it. Although the networks they used are certainly capable of displaying non-reactive behaviours, it is not clear whether the task in question could not equally well have been performed by a feed-forward reactive network. Until they are used to evolve controllers on a task that is non-reactive by nature, judgement as to the evolutionary abilities of these noisy networks to evolve such behaviours must be withheld.

Even simpler recurrent networks were used in the experiments described in chapter 5 (see also (Jakobi 1998)). As a Khepera descended the first arm of a T-maze, it received a light-signal from either the left or the right. In order to perform the behaviour correctly, the robot had to 'remember' from which side the light signal came and turn down the appropriate arm of the maze at the T-junction. The networks that were evolved to perform this task consisted almost entirely of very simple binary-valued neurons that were either 'on' or 'off'. The exceptions were the neurons governing the motors which operated according to a linear threshold function. To calculate the output O_j of the j_{th} neuron, its input activity A_j was first calculated according to the simple weighted sum of equation 2.1

$$A_j = \sum O_i w_{ij} + I_j$$

where O_i was the output from the i_{th} neuron, w_{ij} was the weight on the connection from the i_{th} neuron to the j_{th} neuron, and I_j was any external input to the j_{th} neuron from outside the network. After this, if the j_{th} unit was *not* a motor neuron then its output O_j was calculated according to the function

$$O_j = \begin{cases} 0 & A_j < t_j \\ 1 & A_j \geq t_j \end{cases} \tag{2.3}$$

and if the j_{th} unit *was* a motor neuron then its output O_j was calculated according to the function

$$O_j = \begin{cases} -1 & A_j < t_j - 1 \\ A_j - t_j & t_j - 1 \leq A_j \leq t_j + 1 \\ 1 & A_j > t_j + 1 \end{cases} \tag{2.4}$$

where, in both cases, t_j was a threshold constant associated with the j_{th} neuron.

Networks for dynamically complex non-reactive behaviours

Although conventional networks (such as multi-layered perceptrons) can be made to exhibit non-reactive behaviours, this is not what they were designed for and recurrency alone can be insufficient for problems where the behaviour to be evolved entails complex temporal dynamics. This is because the temporal dynamics of such networks emerge only indirectly from the interactions of parameters such as weights and thresholds. Since weights and thresholds also play other roles in the dynamics of the system these interactions can be extremely epistatic and difficult for evolution to manipulate. For behaviours involving complex temporal dynamics it is often preferable to use types of network where the temporal properties of individual units or the links between them can be accessed by evolution directly. Two different approaches are discussed here.

Beer uses networks of biologically inspired continuous-time neurons originally invented by Hopfield (1984). In these networks, instead of the simple weighted sum of equation 2.1, the input activity A_j of the j_{th} neuron is calculated according to the function

$$\tau_j \dot{A}_j = -A_j + \sum w_{ij} O_i + I_j \tag{2.5}$$

where τ_j is a time constant that affects the rate and extent to which the jth neuron responds to input, O_i is the output from the i_{th} neuron, w_{ij} is the weight on the connection from the i_{th} neuron to the j_{th} neuron, and I_j is any external input to the j_{th} neuron from outside the network. The output O_j of the j_{th} neuron is calculated from the input activity A_j according to the sigmoid function

$$O_j = \left(1 - e^{(t_j - A_j)}\right)^{-1}$$

where t_j is a threshold constant associated with the j_{th} neuron.

This sort of network is well suited to the evolution of behaviours which involve dynamically complex outputs for motor pattern generation. Examples include walking behaviours for multi-legged robots (Beer and Gallagher 1992; Chiel, Beer, Quinn, and Espenschied 1992) (and see chapter 7) and swimming behaviours on simulated lampreys (Ijspeert, Hallam, and Wilshaw 1997). As for behaviours involving dynamically complex input signals, Yamanuchi and Beer (1994) and Beer (1996) have had some success in using this sort of network but it remains to be seen whether this is the best choice for this type of task. Neurons either both react fast and decay fast or react slowly and decay slowly, depending on the time constant. For tasks which involve dynamically complex inputs, however, we often require short-lived stimuli to have long-lasting effects on the internal dynamics. This is of course possible with this sort of network through recurrency, but then the benefits of explicitly parameterising time constants due to a reduction in evolutionary epistasis are lessened.

Chapter 8 describes experiments in which controllers were evolved that allowed a simple panning camera-head to track arbitrarily patterned objects moving against an arbitrarily patterned background. The input signals involved in this behaviour are dynamically complex in that the behaviour may only be achieved through the comparison and integration of different images received over time. The networks used in these experiments were made from binary-valued neurons (either 'on' or 'off') with evolved weights and thresholds, much like those described above for the T-maze experiments of chapter 5. In addition, a time constant was associated with each neuron that specified a period of time that the neuron would remain 'on' after the stimulus that caused it to turn 'on' in the first place had decayed. In this way short-lasting stimuli could very easily have long-lasting effects on the internal dynamics of the system, and these networks were sufficient to evolve successful motion tracking behaviour.

To calculate the output O_j of the j_{th} neuron, its input activity A_j was first calculated from the simple weighted sum of equation 2.1

$$A_j = \sum O_i w_{ij} + I_j$$

where O_i was the output from the i_{th} neuron, w_{ij} was the weight on the connection from the i_{th} neuron to the j_{th} neuron, and I_j was any external input to the j_{th} neuron from outside the network. The output O_j was then calculated according to the function

$$O_j = \begin{cases} 0 & A_j < t_j \\ 1 & A_j \geq t_j \quad or \quad T_{A_j \geq t_j} < \tau j \end{cases} \tag{2.6}$$

where $T_{A_j \geq t_j}$ was the elapsed time since $A_j \geq t_j$ was last true, τj was a time constant associated with the j_{th} neuron and t_j was a threshold constant associated with the j_{th} neuron.

For behaviours which involve complex input, output and internal dynamics, perhaps the best type of network would be a combination of the biologically inspired neurons favoured by Beer and those used in the motion-tracking experiments of chapter 8. The time constants associated with the biologically inspired neurons control both the rate of activation and the rate of decay. The time constants associated with the neurons of the tracking experiments control only the rate of decay. The beneficial properties of both of these approaches could be combined by providing each neuron with two time constants: one for the rate of activation and one for the rate of decay. Although the addition of an extra time-constant per neuron increase the dimensionality of the search space, it also presents evolution with a far easier task in manipulating the dynamics of the evolving systems.

2.2.2 Evolving brain and body

As well as the dynamics of the controller, the physical dynamics of the robot within its environment also play a major role in determining the overall behaviour of the whole system (Smithers 1994; Beer 1995a). The shape, number and arrangement of sensors, for instance, control what features of the environment can affect the inputs of the controller and what features, more importantly perhaps, cannot. On the output side, the physical dynamics of the actuators determine the set of possible actions controllers may use to perform the behaviour with. For these sorts of reasons several researchers have attempted to evolve features of the physical dynamics of the system at the same time as evolving the controller: usually by encoding both on the same genotype. The results are as yet thin on the ground. However, there are a few notable successes where this sort of approach has led to the evolution of behaviours that are unlikely to have come about in any other way.

Evolving the physical properties of an agent in its environment is most easily done when both the agent and the environment exist solely in a simulation. This is for the obvious reason that a physical artifact does not have to be actually realised at each fitness evaluation. Sims (Sims 1994b; Sims 1994a) created a complex virtual world with a 'life-like' physics and evolved the bodies and brains of virtual creatures to perform a variety of different behaviours including swimming, jumping, and competing with other creatures for the possession of a green cube. His motivation for the work was to create interesting and engaging computer graphics but his results certainly provide a vivid demonstration of what is possible if the full physical manifestation of an agent can be put under evolutionary control.

Evolving the physical properties of physical robots in the real world presents much more difficulty. Although there have been a few attempts to evolve a robot's actuators (Lund, Hallam, and Lee 1997; Fukuda 1989), the only real successes I am aware of concern the evolution of a robot's sensor morphology. This was first done by Harvey, Husbands, and Cliff (1994) where controllers were evolved for a visually guided robot that could perform a simple shape discrimination task. Each input to the controller was proportional to the grey-level value of a visual field: a sub-sampled circular region of the camera image whose size and position were under evolutionary control. As well as encoding the information necessary to develop a neural network, each genotype also encoded the position and size of a variable number of these visual fields. The networks that finally evolved to successfully perform the task depended in a fundamental way on the evolved geometry of the inputs as well as the dynamics of the controller. This technique of evolving visual

morphology alongside the controller turns out to be extremely powerful and plays a central role in the experiments of chapters 6 and 8.

2.3 How to encode it?
Encoding schemes for evolutionary robotics

Having decided on the type of artificial neural networks one wants to evolve, the next thing to do is to decide on the process, known generically as the encoding scheme, by which genotypes (the robot-controller 'DNA') get turned into phenotypes (the neural networks). This is important, since the right decision here can mean the difference between success and failure. If we have intuitions about what potential solutions to a problem might look like, then the encoding scheme can be used to bias the evolutionary search to areas of phenotype space that are rich in these potential solutions. If used carelessly it can too easily do the opposite. As an example, consider a robotics task for which successful neural networks are likely to be symmetrical. If the encoding scheme is such that genotype space maps uniformly onto phenotype space, then evolution may take an unacceptably long time simply because symmetrical neural networks make up an extremely small proportion of the space of all neural networks. However, if the encoding scheme is biased or restricted such that the proportion of all genotypes that map onto symmetrical networks is massively increased, then much more of the evolutionary effort will be spent producing and evaluating symmetrical networks, and if our intuitions are correct, we are likely to find a solution a lot quicker.

Using the encoding scheme to build domain knowledge and bias into evolutionary search is an invaluable, and some would say unavoidable, part of the evolutionary robotics process. However, researchers in the field are far from an accord as to how exactly this should be done. One of the most basic disagreements is over whether the encoding scheme should be used to just bias the properties of possible phenotypes so that random genotypes are *likely* to code for phenotypes that fit with our intuitions, or whether it should be used to restrict the properties of possible phenotypes so that random genotypes *only* code for phenotypes that fit with our intuitions. Both options have their advantages and disadvantages. If the former option is employed then this will speed up the time taken to find a solution if our intuitions are correct, and we may potentially still find a solution even if our intuitions are wrong. The latter option means that if our intuitions are correct then evolution is likely to find a solution even quicker than with the former option since it will *only* be searching through phenotypes that fit with these intuitions. However, if our intuitions are wrong, then this option means that evolution can never find a solution.

I think encoding schemes should restrict rather than bias the phenotypes they code for. My arguments for why are pragmatic but quite subtle (Jakobi 1996a; Jakobi 1996b) . Evolving complex robot behaviours is so difficult using current techniques that often the only way that solutions can be found within an acceptable time-period is if enough well-founded intuitions and heuristics are built into the encoding scheme. So if solutions can *only* be found within an acceptable time-period if sufficient well-founded heuristics are employed then there is no point in making allowances for when these heuristics are *not* well-founded. Another way of understanding this is to realise that by using an encoding scheme that biases evolution in favour of phenotypes that display a certain property, evolution is biased against those phenotypes that do not display that property. This makes it likely that if the problem is hard enough that we need to apply our intuitions, then if our intuitions

are wrong, evolution will not find a solution within an acceptable time using an encoding scheme that is, in effect, biased *against* finding solutions. There is therefore no point in using an encoding scheme that allows genotypes to code for phenotypes that do not fit with our intuitions. If we think that successful networks will be a certain shape or size or display some other property, then the encoding scheme should restrict the possible phenotypes that may be encoded to those that are of that shape or size or display that property.

The one exception to this is when we want to apply an intuition to a problem, but the intuition is not an all or nothing thing but a matter of degree. An example of this might be when we are confident that successful networks will display some repeated structure, but we are not sure exactly how much and we therefore want evolution to decide. In this case, restricting phenotypes to displaying a certain fixed amount of repeated structure misses the point, and I would advocate using an encoding scheme such as Gruau's cellular encoding (Gruau 1994) where the exploration of the amount of repeated structure may be very tightly controlled within well-defined limits. I would not advocate, for practical purposes, the use of any of the more biologically inspired encoding schemes modelled on natural development (Dellaert and Beer 1994; Cliff 1994; Nolfi and Parisi 1995a; Jakobi 1996c; Kitano 1995; Dellaert and Beer 1996; Vaario 1993; Kodjabachian and Meyer 1994). These are interesting from a biological point of view, but do not come close to the more abstract schemes described below in terms of their efficiency to evolve robot controllers.

2.3.1 Encoding schemes for networks of fixed size

Neural networks of fixed size, or more importantly a fixed number of parameters, present none of the encoding problems associated with networks whose size is under evolutionary control (see below). It is therefore sufficient in almost all cases to use a simple one-to-one mapping from genotype to phenotype. Such a one-to-one mapping is commonly referred to as a 'direct' encoding scheme, and has been used by several researchers to successfully evolve neural networks of fixed size for the control of robots (Floreano and Mondada 1996a; Nolfi and Parisi 1995b; Gallagher, Beer, Espenschied, and Quinn 1996). In its most common form, each genotype is made up of a string of characters or numbers, and each parameter of the network has its own position on the genotype. Thus the first 20 characters might code for the parameters of neuron 1 and its links to other neurons in the network, and the second 20, neuron 2 and its links and so on.

The space of possible phenotypes that a direct encoding scheme can produce may be easily restricted by allowing a single region on the genotype to code for more than one set of network parameters[2]. For instance we may impose symmetry by encoding the parameters for only half a neural network and create the other half through reflection. Repeated structure on a larger scale may be imposed by encoding the parameters for a single sub-network that is then repeated several times throughout the network as a whole. Techniques like these are used to create the symmetrical controllers that allow the Khepera to perform the T-maze task of chapter 5, and the 6-fold symmetrical hexapod controllers of (Gallagher, Beer, Espenschied, and Quinn 1996).

There are also ways of keeping the number of neurons constant but allowing the number of links between neurons to vary without straying too far from the simplicity of direct encoding schemes. The scheme used for the experiments of chapter 5 constitute a nice demonstration of

[2]It is not so easy to bias, rather than restrict, the set of possible genotypes using a direct encoding scheme.

this. The 10 neurons that make up each network are numbered 0 to 9, and every neuron has three possible links associated with it. Each link indexes the neuron it connects to with a number between 0 and 16, and if a link tries to connect to a neuron with a number greater than 9 then it will just fail to connect. Thus the number of links connecting to each neuron is under evolutionary control with a maximum of 3 and a minimum of 0. The length of the genome, however, remains fixed, and the problems of varying length genotypes described below are avoided.

2.3.2 Encoding schemes for networks that vary in size

If we want to do open-ended evolution where the complexity of a solution is not known beforehand then we need to use an encoding scheme that allows the neural networks that it encodes to vary in size. Such a scheme must enable neurons to be added or subtracted from a network without overly disrupting its overall shape and form. Also, if we are using crossover, then this must be able to produce viable offspring when performed on two genotypes that code for networks of different size. This is actually quite hard to achieve since variation in genotype length means variation in the relative positions of each neuron's coding section on the genotype. Thus a simple direct encoding scheme which specifies the links between neurons by their coding positions on the genotype (e.g. neuron 4 connects to neuron 2) will be massively disrupted by changes in genotype length. Cliff, Harvey and Husbands suggest as a solution using a mixture of absolute and relative addressing (Cliff, Harvey, and Husbands 1993), but this is still subject to similar problems. Others have tried to duck the issue altogether by introducing mechanisms that allow the size of the network to vary while keeping the size of the genotype fixed (Jakobi 1994; Cliff 1994). However, these schemes come with their own limitations not least of which is that they set an upper limit to the amount a neural network can grow which may not always be desirable. The two encoding schemes described below are the only two I am aware of which both allow variable length genotypes and are robust to genetic operators.

Cellular Encoding

Gruau has put forward an encoding scheme for the evolution of neural networks called 'cellular encoding'. This has been applied successfully to the evolution of controllers for a simulated hexapod robot (Gruau 1995) and a real octopod robot (Gruau 1997). For details of how it works and how it is able to evolve solutions to many of the more abstract and traditional neural network problems such as the n-bit parity and symmetry problems see (Gruau 1994). The power of cellular encoding derives from the fact that it uses Lisp-like tree structures of graph rewriting rules as genotypes to develop the size, connectivity and parameter values of arbitrarily large graphs of connected nodes: the neural network phenotypes. Starting from an initial graph consisting of a single neuron connecting inputs to outputs, development proceeds by applying successive graph transformations (taken in order from the top of the genotype tree downwards) until all the 'leaves' of the tree are reached and the neural network is complete. To get at least some idea of this, figure 2.4 shows some examples of the sort of graph transformations that are used and what happens when they are applied to the initial graph. For a full picture, the reader is referred to (Gruau 1994).

The use of Lisp-like tree structures as genotypes means that the scheme is extremely robust with respect to different sized genotypes. Nodes may be added or subtracted by adding or sub-

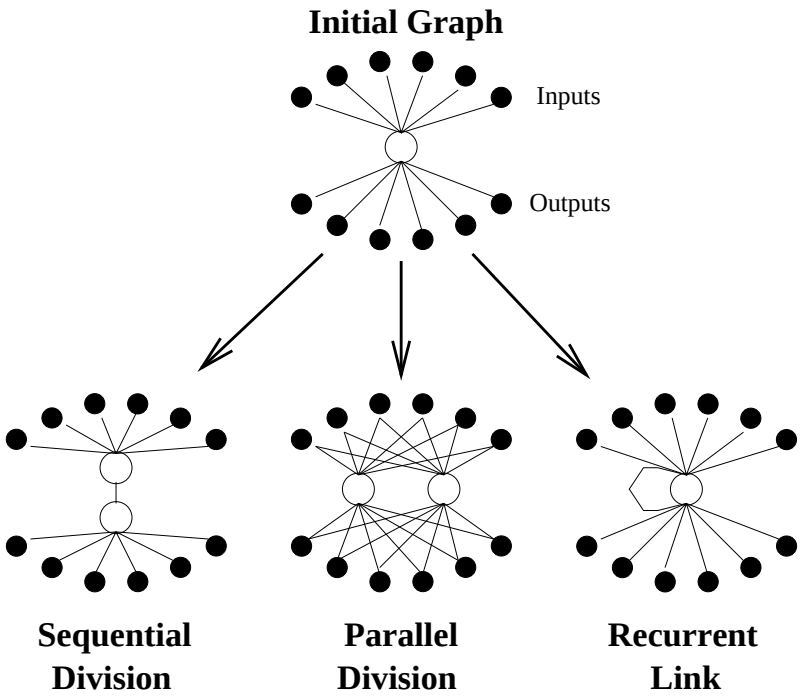

Figure 2.4: *This shows three examples of the sort of graph transformations that are used in cellular encoding and what happens when they are applied to the initial graph.*

tracting terminals, and crossover may be easily implemented between trees of different sizes by exchanging sub-trees. Gruau has shown how cellular encoding is able to generate any size and shape of network, and by careful selection of the types of graph transformations that form the genotype trees, network modularity and repeated structure may be easily and naturally imposed on evolving networks. The scheme is also the only scheme available that is capable of biasing evolving networks towards say, repeated structure, in a controlled and efficient enough way to be used by artificial evolution in practice. This can be useful for problems where it is known that successful networks will display repeated structure, but it is not known beforehand how much.

The only problem with cellular encoding is that it is complicated and unintuitive, and a proper implementation requires a large computer program that will slow the process of artificial evolution down. Large computer programs also mean bugs. Unless the problem at hand really does require an exploration of the amount or degree of repeated structure, therefore, I advocate the use of the much simpler and intuitive spatially determined encoding described below.

Spatially Determined Encoding

In a normal direct encoding scheme, the genotypic code for a neuron and its links has a particular location on the genotype. The code for each connection specifies another location on the genotype (either through relative or absolute addressing) where the code for the target neuron of that particular connection can be found. As stated above, this works fine except when genotypes are allowed to vary in size. If this occurs, then the locations on the genotype of the code for each neuron can change, and this can have massive effects on the shape and functionality of the network.

This problem can be eleviated by genetically specifying a particular location in an independent

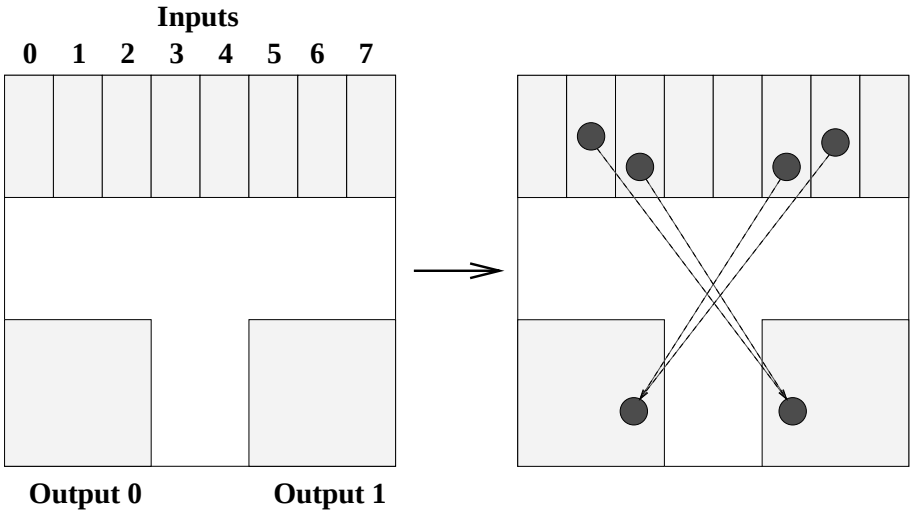

Figure 2.5: *A simple network generated using the spatial encoding scheme. The left-hand diagram shown how the developmental space can be divide up into input regions, output regions, and 'hidden' unit regions. The eight inputs in this example might correspond to the IR sensors of a Khepera robot and the outputs to its two motors. The network on the right, typical of those that underly obstacle avoidance in the Khepera, is generated through reflection from the genotypic specification of three neurons.*

developmental space for each neuron over and above its location on the genotype. If the links between neurons are then genetically specified using these new locations, it does not matter where the code for a neuron is situated on the genotype: its location within the developmental space remains the same and thus so do its links to and from other neurons. Several encoding schemes have been invented that use this idea (Cliff 1994; Nolfi and Parisi 1995a; Husbands, Harvey, Cliff, and Miller 1994), although for the most part they take their inspiration from biology and are unnecessarily complex. For problems in which the degree of repeated structure of evolving networks is fixed, I advocate the simple spatially determined encoding scheme first used in (Jakobi 1998). We will now look in detail at how a genotype develops into a neural network using such a spatially determined encoding scheme.

The left-hand-side of figure 2.5 shows a two-dimensional example of a typical developmental space in which development can take place. For each possible network input and output, a region of this space is specified. Development begins by plotting the position of each neuron into the space according to their genetically specified locations. Depending on which region each neuron is plotted into it is identified as an input neuron, an output neuron or a hidden neuron.

After plotting the locations of all of the neurons in the network, the connectivity of the network is then determined. As with a direct encoding scheme, a large part of the genetic code for each neuron specifies a certain number of links to or from other neurons. The code for each link specifies a target location within the developmental space. If there are any neurons located within a predefined range of this target location, then the nearest one of these is identified as the neuron which the link connects to. If there happen to be no neurons situated within this range then the link fails to connect. The right-hand-side of figure 2.5 shows a very simple, symmetrical network developed from the genetic code for three neurons by a process of reflection.

Spatially determined encoding is simple, in that two paragraphs suffices to completely de-

scribe it, and yet it suffers none of the problems that direct encoding schemes suffer when the genotype is allowed to vary in size. Crossover between genotypes of different lengths may also be easily achieved through reference to the developmental space. Instead of randomly determining a crossover point on the genotype, a crossover line is determined on the developmental space (a horizontal line in the space of figure 2.5). If two parent genotypes of different length are selected to form a single offspring genotype, then the sections of parent 1 that code for neurons which are located in the developmental space on one side of the crossover line are joined to those sections of parent 2 that code for neurons which are located on the other side to form the offspring[3]. In this way, the crossover operation again has nothing to do with the spatial organization of the genotypes, and therefore does not run into any of the problems associated with crossover between genotypes of different lengths.

This encoding scheme is a powerful tool for evolving complex neural networks. It can also be easily extended so that other position-defined properties of a controller, such as its sensor morphology, may be encoded on the same genome. The experiments of chapters 6 and 8 both involve versions of spatially determined encoding.

2.4 How to evolve it?
Genetic algorithms and genetic operators for evolutionary robotics

Apart from the encoding scheme, there are several other processes essential to an evolutionary robotics system that work directly on or with genotypes. These include the genetic algorithm itself, which is responsible for the high-level allocation of tasks between all the other processes of the system, and the genetic operators which are responsible for generating new genotypes from old. Both of these types of processes are examined below. Before entering into a discussion of exactly what form these processes should take, however, we first need to define exactly what genotypes themselves are to consist of.

Apart from Gruau's cellular encoding, all the encoding schemes discussed in section 2.3 naturally operate with strings of values as genotypes. Cellular encoding is an exception to this since it operates with Lisp-like tree structures instead. Since there are only certain circumstances in which cellular encoding is advocated, however, the functions proposed below are described in terms of operations upon strings of values. Equivalent functions could be implemented for Lisp-like tree structures, but what these might look like is left as an exercise for the reader. Other alternatives to using simple strings include using genotypes made from two strings (diploid) rather than one (haploid). Results in (Calabretta, Galbiati, Nolfi, and Parisi 1996) suggest that this may increase the ability of evolving individuals to 'buffer environmental change'. If the environment is not constantly changing, however, then the increase in complexity to the setup as a whole is probably not justified.

Specifying that genotypes are made up of strings of values still begs the question of how these values should be represented. Although many researchers (Holland 1975; Goldberg 1989b; Goldberg 1989a) prefer to use genotypes of 1's and 0's where each parameter is binary-encoded

[3]Note that this can easily be done without having to develop each parent genotype into a network. If the third parameter of the code for a neuron corresponds to its vertical position in the developmental space, then crossover just involves travelling down each genome, looking at each third parameter and deciding by its value which of the offspring genotypes the code for that neuron belongs to.

by a certain number of bits, others prefer to use real numbers (Back, Hoffmeister, and Schwefel 1991; Back and Schwefel 1993). I advocate using genotypes made from strings of real numbers for the following reasons:

- The probability distribution of the ways in which parameters can change under mutation may be tightly controlled. If a binary encoding is used then this probability distribution is always the same: the effects of a mutation to the k'th bit being 2^k time more than the effects of mutation to the 0'th bit. We have no reason for thinking this is a good distribution.
- Unless some sort of Gray encoding is used (Salomon 1996) then the possible effects of single mutations on a particular binary number will vary, depending on the value of that number. For example it is not always possible to increment by one the value of a binary number through a single-point mutation. The same is of course not true of real-number encodings.
- Genotypes made from real numbers are easier to use and write computer programs with than binary strings.

Possible arguments *for* using binary values and *against* using real-numbered values might be couched in terms of the schema theorem and building block hypothesis (Goldberg 1989b), both of which have been put forward as attempts to explain how and why genetic algorithms can be so efficient in classical optimization problems. However, if a genetic algorithm is run according to SAGA principles (Harvey 1992) so that the population is highly converged for the majority of the evolutionary run (see below), then neither the schema theorem nor the building block hypothesis have much purchase.

2.4.1 Genetic algorithms for evolutionary robotics

Figure 2.1 shows a diagram of a simple genetic algorithm: the top-level function that controls and coordinates the evolutionary process as a whole. Specifically, the genetic algorithm (GA) is responsible for the process by which genotypes are selected to act as parents for the creation of offspring, and the process by which new offspring genotypes are introduced back into the system to replace the old. All the other processes that are part of an evolutionary robotics system such as the evaluation process and the reproduction process can be ported from one type of genetic algorithm to another.

Genetic algorithms come in various forms: generational, steady state, distributed and so on, and a good introduction can be found in (Mitchell 1996). However, as the 'no free lunch' theorems make clear (Wolpert 1995), there is no such thing as the best GA for *all* problems, but only the best GA for a particular problem. Although certain types of GA have been shown to perform better than others on certain types of fitness landscapes (De Jong 1975), it is not clear how these landscapes relate to those involved in a typical evolutionary robotics scenario. Until this work is done, therefore, or a comparison of various GAs is made on some test-suit of evolutionary robotics tasks, it is not clear what is the best overall type of GA for evolutionary robotics. Given an arbitrary decision, I tend to err on the side of simplicity (see below). There are however a few general things one can say about the sort of genetic algorithms one should use, given the specific problems one runs into when using artificial evolution for the creation of robot controllers.

One feature of robotics that sets it apart from many other domains to which artificial evolution has been employed is noise. If we are working with real robots, or simulations that behave like real robots, then any evaluation function based on some empirical measurement of a controller's ability to perform a task will be noisy. This means that comparisons between different controllers will often yield different results on different occasions, and those individuals that receive the highest fitness values may not always be the fittest. If a genetic algorithm is to be used in evolutionary robotics, therefore, then it must be as robust as possible to noise on the fitness test. In particular the way in which genotypes are picked as parents for the creation of new offspring must be as robust to noise as possible. One way of doing this is to evaluate each individual a number of times, instead of just once, and take an average score as the fitness value. If resources are tight, then various statistical techniques can be used which allocate the number of trials per genotype in the most efficient way (Aizawa and Wah 1994; Fitzpatrick and Grefenstette 1988). Keeping the selection pressure low to prevent the population from dashing off in the wrong direction through fitness space also helps, and hard elitist strategies whereby the fittest is copied from generation to generation without re-evaluation is not a good idea. Apart from these few hints, though, it is not obvious which selection strategies are more robust to noise than others.

One interesting avenue of research on this topic concerns an effect reported in (Eigen 1987; Huynen and Hogeweg 1994) and adapted for engineering purposes by Thompson(Thompson 1996a; Thompson 1995b). Given the choice between two peaks of equal height in the fitness landscape, evolution seems to favour the peak whose fitness is the more robust to mutation. Moreover Thompson has shown that this effect only occurs if certain selection strategies such as rank selection, and fitness proportional selection are used. The effect seems to be less noticeable if truncation selection, for instance, is employed. The amount which fitness values vary across offspring events in response to mutation is equivalent to the amount fitness values vary across offspring events in response to noise. If evolution favours those places in the fitness landscape which vary less in response to mutation then the same effect may mean that it will also favour those places in the fitness landscape which vary less in response to noise. This is only a hypothesis, but it seems reason enough, given an otherwise arbitrary decision, to use the selection strategies that Thompson has found pronounce this lesser-known feature of evolution to the fullest.

Taking all of this into account, I advocate the use of a simple, easy to program genetic algorithm such as the basic generational example of section 2.1. The selection should be rank-based and afford even the least fit members of the population a chance to participate in offspring events. All the examples of chapters 5, 6, 7 and 8 use genetic algorithms of this type.

2.4.2 Genetic operators for evolutionary robotics

The genetic operators are responsible for the way in which new offspring genotypes derive from old parent genotypes. Some of the more commonly used include the crossover operator that combines the genetic material of two parent genotypes to produce offspring, the mutation operator that introduces random mutations into the genotype, and the translocation operator that randomly relocates segments of genetic code elsewhere in the genotype (Mitchell 1996). Since the genetic operators are responsible for how similar offspring genotypes are to their parents, they play a large role (in concert with the selection pressure) in determining the degree of genetic diversity

within the population, and the rate at which it converges from its initial random state. This in turn determines the speed and operational characteristics of the evolutionary process.

For many who use genetic algorithms on more traditional optimization problems, the process of artificial evolution occurs as an initially random population converges upon a solution, gradually decreasing the genetic diversity until an equilibrium is reached. At this point, it is assumed that no significant change will occur and the process for all practical purposes is finished. If genotypes are of fixed length then the same can be assumed for most evolutionary robotics problems: at some point the population will converge, the rate at which fitness increases will level off, and the run is over. In this sort of scenario, evolutionary robotics problems may be treated as noisy optimization problems. and mutation, crossover and other operators may be applied using conventional genetic algorithm techniques (Goldberg 1989b).

The situation is made more complex if genotypes are allowed to grow and vary in length. This is because there is always the possibility that significant change can occur *after* the rate of genetic convergence has stabilized. To take advantage of this, Harvey (1992) developed the principles of Species Adaptive Genetic Algorithms (SAGA). If we want to do open-ended evolution of arbitrary complexity with variable length genotypes, he suggests, then we should allow the evolutionary process to continue running long after the rate of genotypic convergence has stabilized. In the natural world, after all, evolution occurs as a dynamic equilibrium that adapts to environmental pressures in an open-ended way rather than a limited and finite search process with a start and a finish. Perhaps it is only after this initial convergence phase, therefore, that the real business of open-ended artificial evolution may begin.

There are a variety of ways in which genotypes can be allowed to change in length under evolutionary control. Probably the simplest is to employ operators that just add or delete genetic material with a small chance at each offspring event. Another method might be to allow crossover to occur at different points on each parent genotypes, thus producing two offspring genotypes of different lengths. Whichever method is used, however, no more than a slight change in length should be allowed to occur at each offspring event. Although it is important that there is sufficient generation and evaluation of new genetic material after the rate of convergence has stabilized, too much will overpower the selection pressure and reduce the adaptive abilities of the evolutionary processes to those of random search.

In the context of a converged population, the role of the crossover operator is unclear. It may on occasion combine useful ‘building blocks’ from two parent genotypes in the same offspring genotype, but the fact that all individuals within the population are very similar means that this will not occur often and will not therefore play a major role. There are arguments that crossover lessens the effects of Muller’s Ratchet (Nowak and Schuster 1989) but again it is hard to see how this makes it crucial to the evolutionary process. It may be that by incorporating some sort of sexual selection (Todd and Miller 1993; Todd 1996) as well as fitness proportional selection, crossover may be made to earn its keep. Until this has been shown, however, thought must be given to whether the extra unpredictable mutation effects produced by this operator are worth the gain.

If we decide against using crossover, then we must rely on random mutation in one of its many

forms[4] to maintain the genotypic variation necessary for ongoing open-ended evolution. However, the question of how to apply mutations when dealing with variable length genotypes again presents us with a dilemma. Either mutation is fixed at a particular rate per genotype or it is fixed at a particular rate per unit of genotype. Both have their problems. To illustrate this, let us consider a variable length genotype that can be divided up into a variable number of 'genes', depending on its length, each coding for a neuron or group of neurons and consisting of a certain fixed number of values. If the mutation rate is fixed at a particular rate per genotype, then as the length of the genotype increases, the effective mutation rate per gene decreases in proportion to the reciprocal. Thus the more complex the neural network encoded on the genome, the slower new complexity evolves until eventually, in practical terms, evolution grinds to a halt. If the mutation rate is fixed at a particular rate per gene then, as the genotype grows in length, the effective mutation rate per genotype increase until we are faced with a different problem: that of the error catastrophe (Nowak and Schuster 1989). This occurs when the mutation rate is so high that fit traits and individuals are lost to mutation before they have a chance to spread through and establish themselves in the population. Thus mutation overcomes the forces of selection, and the rate at which new complexity evolves again drops to zero.

One answer to this dilemma is to protect 'fit' genes from the deleterious effects of major mutations in some way. The existing work on adaptive mutation rates is a step in this direction (Smith and Fogarty 1997), however here we consider the circumstances under which 'fit' genes might be protected from major mutation completely. Once it has been established that a gene contributes to the fitness of genotypes that contain it, then it will usually continue to contribute to the fitness of such genotypes from generation to generation. When this will not be true is if the fitness peak that the population occupies due to the gene is a local maximum, and later on in the run evolution finds a higher maximum elsewhere whose fitness the gene does not contribute to. This is not, however, what we observe happening in an open-ended evolutionary robotics performed on a static fitness landscape according to S.A.G.A. principles. In this case, populations are converged, and once evolution has come up with a way of producing a particular behaviour it tends to stick to it. Later on, with growth of the genotype, it may evolve to produce other behaviours as well. Thus if a gene is beneficial (i.e. plays a significant role in increasing the fitness of genotypes that contain it) then we can expect it to stay in the population indefinitely, and protect it from major mutations without negative consequences to the evolutionary process as a whole.

If an automatic process can be found which 'mutation-locks' beneficial genes of this type then this would offer a possible solution to the problem of applying mutation in an open-ended evolutionary robotics scenario. This is because, with the correct genotypic growth rate, the number of *un-locked* genes can be kept more or less constant as genes lock automatically. Therefore a mutation rate per gene, which maintains the degree to which new complexity is explored, will result in an effective mutation rate per genotype that is also constant, thus avoiding the error threshold.

The experiments of chapter 8 employed an automatic mutation-locking process inspired from the way in which some genes on a real chromosome are more likely to undergo mutation during reproduction than others due to their evolutionary age. This process works as follows. Associated

[4] I am counting any operator that has a random effect on a single genotype, such as translocation, as a form of mutation.

with each gene on the genotype is a number that represents the gene's 'age'. Initially, all genes are aged zero. At each offspring event, the age of every gene that is copied unscathed from parent to offspring is increased by 1, and the age of those genes that undergo major mutation are reset to zero. With this setup, the only way in which a particular gene can achieve an old age is if genotypes that contain unmutated copies of it are constantly selected to act as parents. If genes that contain mutated copies are equally likely to be selected, then after a certain amount of time we can expect the gene in its unmutated form to disappear from the population due to the effects of genetic drift. After a gene reaches a certain age, therefore, we can be reasonably confident that it is beneficial and protect it from any further major mutation. This age can be arrived at empirically by running the genetic algorithm on a neutral fitness landscape for several thousand generations and observing the maximum age achieved by a gene in that time. When run on the real fitness landscape, only beneficial genes will survive for significantly longer than this maximum age.

2.5 How to evaluate it?
Fitness function issues for evolutionary robotics

A typical evolutionary robotics run will involve the constant and repetitive evaluation of hundreds upon thousands of robot controllers. Since these evaluations involve controllers' ability to behave in a certain way or perform a certain task on a *real* robot, this process is far from a trivial matter; a choice has to be made between whether evaluations are to be done using the real robots themselves or using simulations of the real robots, and both of these alternatives have their problems. Evaluation on real robots, for example, has to be performed in real time which may take prohibitively long. Evaluation in simulation, on the other hand, means that evolved controllers may not work when transferred into reality unless the simulation is prohibitively complex. How to overcome these problems is one of the major challenges faced by evolutionary robotics at the moment (Mataric and Cliff 1996) and one of the main issues tackled by this thesis. Chapter 3.3 introduces new ways of thinking about and building fast-running easy-to-design minimal simulations for evaluating real robot controllers.

Before getting onto this question of whether an evaluation is best performed in simulation or in reality, however, we first take a look at what such an evaluation might consist of. The rest of this section discusses the best way to go about designing and formulating fitness functions for the automatic prescription of fitness values.

Typically in evolutionary robotics, the fitness of evolving controllers is evaluated by running them on either a real or simulated robot (see next chapter) and using an automatic fitness function[5] to judge their ability to perform a task or behave in a certain way. This fitness function operates upon measurements of features of the robot and its environment and the way in which they change over the course of the trial period. Of course, which of the features of the robot and its environment can be measured and used by the fitness function depends on the robotics setup in question: it is one of the advantages of performing evaluations in simulations that practically every one of these features is at our disposal. There is, as yet, no principled way of designing a fitness function for evolutionary robotics, and as several researchers have pointed out (Floreano and Mondada 1996a; Mataric and Cliff 1996), it can be a laborious trial-and-error process. There are, however, a few

[5] Although fitness can also be prescribed by hand. See (Gruau 1997) for an example of this.

design criteria which can aid the evolution of the behaviour we are after:

- The only way in which evolving controllers should be able to get a high score is if they successfully display the behaviour we are after. Careful attention must be paid to this: evolution is not fussy and will exploit any loop-holes it is presented with.
- At all stages during the evolutionary process, the extra complexity needed to evolve in order that fitness can increase should be as small as possible. This is akin to saying that if a behaviour can be broken down into sub-behaviours, then the fitness function should incrementally reward the evolution of each sub-behaviour.
- The value returned by the fitness function must be as accurate a reflection of the underlying fitness of the controller as possible. For this reason, the measured features of the environment that we choose to base the fitness function upon should be those that are the most reliable and least noisy.
- If all we are after is a robot controller that displays a particular behaviour, then it does not matter how specific the fitness function is. If, however, we want to explore a space of possible robot behaviours, then the fitness function must be as un-biased as possible, and we must take great care that it rewards all the different ways there are of performing the particular behaviour equally.

Examples of what fitness functions look like in practice are provided in chapters 5, 6, 7 and 8.

2.6 Here endeth the lesson

This chapter has provided a crash course in state-of-the-art Evolutionary Robotics. It is hoped that the reader who wants to evolve a particular robot behaviour will now have a good idea of the type of neural networks they should use, how to encode them onto a genotype, and the best genetic algorithms and operators to use. However, although section 2.5 offers some ideas about how to go about designing a suitable fitness function, it did not go into details of how this fitness function should be applied to evolving controllers in practice. In particular, the question of whether evolving controllers should be evaluated in simulation or reality has still to be answered.

One of the major aims of this thesis is to offer an answer to this question. The next chapter introduces new ways of thinking about and building fast-running easy-to-design minimal simulations that can be used to evaluate evolving controllers for real robots. A formal treatment of the theory behind these minimal simulations is then presented in chapter 4. Chapters 5, 6, 7 and 8 provide details of experiments in which these minimal simulations were used, in conjunction with many of the techniques explained in this tutorial chapter, to successfully evolve controllers for real robots.

Chapter 3

Minimal simulations I: General Theory and Methodology

The artificial evolution of controllers typically involves the constant and repetitive testing of hundreds upon thousands of individuals as to their ability to behave in a certain way or perform a certain task. In the case of real robots this testing procedure is far from a trivial matter and (with the exception of certain hybrid approaches (Thompson 1995a; Nolfi, Floreano, Miglino, and Mondada 1994)) can be done in only one of two ways: controllers must either be evaluated on real robots in the real world, or they must be evaluated in simulations of real robots in the real world. Both of these approaches have their problems.

As Mataric and Cliff (1996) point out, the evaluation of controllers on real robots must be done in real time, and this makes the entire evolutionary process prohibitively slow. As an example, they cite the evolution of collision-free navigation on a Khepera robot, which in the experiments reported in (Floreano and Mondada 1994) took a total of 65 hours (100 generations at 39 minutes a generation) to evolve[1]; it is hard to see how this approach can scale up, if the behaviours we are after require thousands or even millions of generations. But even if we are resigned to an evolutionary process that takes years rather than days, then there are different problems that must be faced. The process must be automated, for instance. This begs questions about how data is to be collected for fitness evaluations, how the robot is to be returned to its starting position at the end of each fitness trial without human intervention and so on. Power must also be supplied continuously to robots in situations where batteries have limited life-spans and tethering by a permanent power lead is not always possible. And machines break down, especially under the sort of continuous random battering that the real-world evaluation approach advocates. Clearly the alternative simulation approach would be preferable since it avoids all these problems. It can also run at faster than real time.

As has been shown by several experimenters (Jakobi, Husbands, and Harvey 1995; Beer and Gallagher 1992; Miglino, Lund, and Nolfi 1995), it *is* possible to evolve controllers in simulation for a real robot. Now that this is no longer in doubt the question becomes one of whether the technique will scale up. Mataric and Cliff (1996) (and similar points were made earlier in (Husbands and Harvey 1992; Brooks 1992; Harvey and Husbands 1992)) argue that if behavioural transfer-

[1] The shape-discrimination behaviour evolved in (Harvey, Husbands, and Cliff 1994) only took 36 hours to evolve, but this is still of the same order of magnitude.

ence can only be guaranteed when a carefully constructed empirically validated simulation is used, then as robots and the behaviours we want to evolve for them become more complicated, so do the simulations. The level of complexity involved, they argue, would make such simulations:

- so computationally expensive that all speed advantages over real-world evolution are lost.
- so hard to design that the time taken in development outweighs time saved by fast evolution.

Clearly the main challenge for the simulation approach to evolutionary robotics is to invent a general theoretical and methodological framework that enables the easy and cheap construction of fast-running simulators for evolving real-world robot behaviors.

This chapter puts forwards such a theoretical and methodological framework. It starts in section 3.1 with an analysis of what it means to say that a controller transfers across the 'reality gap' from a simulation into reality. This is then used in section 3.2 to investigate the general conditions under which controllers will and will not transfer. Section 3.3 introduces the idea of a minimal simulation - the simplest possible type of simulation capable of evolving robot controllers - and recasts the conditions on controllers for successful transfer within this context. Techniques are then proposed, in section 3.4, for using the evolutionary process itself to force controllers that evolve to perform the desired behaviour within a minimal simulation to also fulfill the conditions for successful transfer into reality. All the various threads are brought together and summarised, in section 3.5, to produce a simple step-by-step guide to building a minimal simulation for evolutionary robotics. Finally, a discussion of the pros and cons of the minimal simulation approach is offered in section 3.6.

3.1 What is this reality gap anyway?

If we are to develop a general methodology for building simulations for evolutionary robotics then the first thing to do is to define what we mean when we say that a controller has transferred from simulation into reality. There are several reasons for doing this, not least of which is that if we do not all have an agreed set of criteria for successful transfer, then there is little point in asking under what conditions these criteria might be fulfilled, since we will all be asking different things. This section sets up a terminological and conceptual framework within which a criterion for successful transfer (and the discussion in the following sections) is stated. While the definitions given below may not be agreed upon by everyone as the *only* way or the *best* way in which to cut the conceptual pie, they should nevertheless be seen as *a* way: and that is all that is needed to ensure that the discussion to come is unambiguous and precise.

The section begins by drawing a line between a controller and its environment in order that we may be quite clear about exactly which features of the real-world situation belong to the controller (which we do not have to simulate) and which features belong to the environment (which we do). It then goes on to ask under what conditions a controller can be said to be performing a particular behaviour within a particular environment and gives two conditions: one on the controller and one on the environment. Finally a definition of successful behavioural transference is stated.

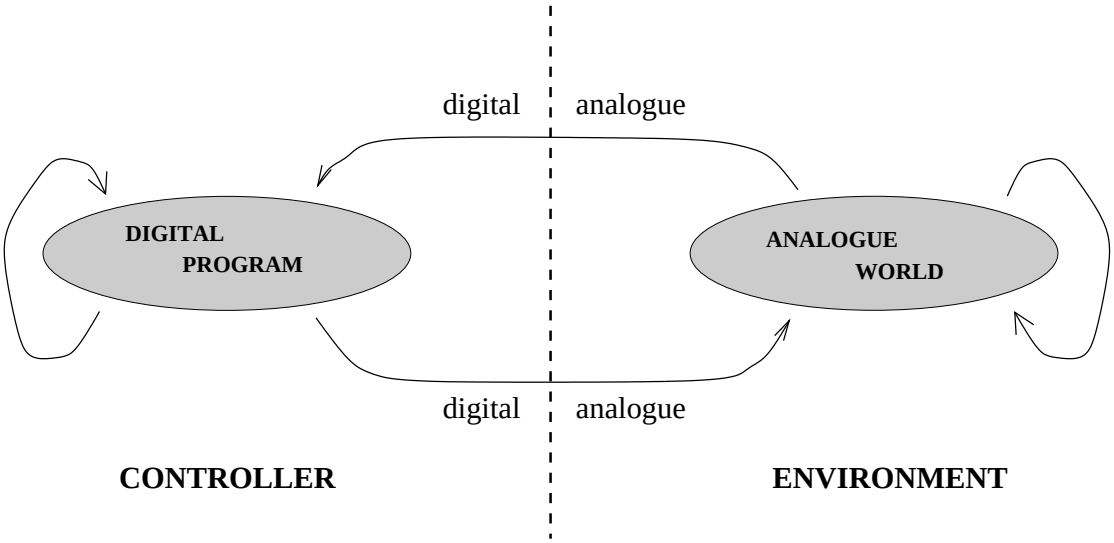

Figure 3.1: *This figure shows where, for the purposes of this paper, the line is drawn between controller and environment.*

3.1.1 Drawing the line between controller and environment

The distinction between controller and environment can be drawn in a number of ways (Smithers 1994; Beer 1995a; Smithers 1997). For the purposes of this paper, the boundary needs to be drawn such that a controller running on a robot and a controller running in a simulation can uncontroversially and simply instantiate two copies of the same dynamical system; if this is not done, then we cannot talk about testing the *same* controller in both simulation and reality.

In each of the four sets of experiments described in chapters 5, 6, 7 and 8, controllers are dynamical systems instantiated by a piece of code running on a computer. The environment is another dynamical system instantiated *either* by a real world dynamical system such as a mobile robot acting in the world *or* by a piece of code running on a computer that simulates such a real world dynamical system. Figure 3.1 shows diagrammatically how the boundary between controller and environment is drawn for the purposes of this paper. The digital signals that are output from the controller undergo some digital to analogue processes within the environment that gives rise (via the motors) to actions in the world, and analogue to digital processes within the environment give rise (via the sensors) to digital signals that are input to the controller. The boundary between controller and environment occurs at those point where signals are more profitably regarded as digital rather than analogue or vice versa. The most important thing to realize about this particular distinction is that the environment contains the robot's sensors and actuators or their simulated equivalent

If the boundary is drawn in this way, the controller can be seen as existing entirely in the digital domain. Provided copies of the controller in simulation and reality are run on computers that use the same rules of computation and run at the same rate, therefore, they can be seen as constituting two copies of the same dynamical system.

3.1.2 What counts as an instance of a behaviour?

Having defined what we mean by a controller, and what we mean by an environment, this subsection examines the conditions under which a controller can be said to perform a particular behaviour within an environment. In the next subsection, we will examine the conditions under which a controller can be said to perform the same behaviour in two different environments: simulation and reality, for example.

The internal state of a controller and the way in which the environment acts upon it are, in general, central to the generation and maintenance of a particular behaviour. It is the controller's output and the effects this has on the state of the environment, however, that determines whether or not, from an observer's point of view, the agent actually performs the behaviour[2]. To illustrate this, consider the behaviour that consists of driving a car down a road. If someone sits in a car and steers it down the road with a blindfold on, then we will still say they are driving the car (blind-folded) down the road. However, if someone receives visual input as if they were driving the car, but they are not in the driver's seat and are not touching the steering wheel, then we will say they are not driving the car down the road. In general, whether or not a controller performs a particular behaviour within a particular environment depends on the way the state of the environment changes in response to controller output rather than the way in which the state of the controller changes in response to input from the environment.

The environment must be capable of supporting the behaviour

The way in which the state of the environment changes is obviously a function of the controller's motor signals, but it is also a function of the environment itself and the ways in which the controller may interact with it. For instance, if a person makes breast-stroke motions with their arms and kicks their legs, then they are producing the correct motor output for swimming. However, it is not until they are immersed in water or something equivalent that they actually *are* swimming[3]. As another example, consider corridor-following behaviour by a small mobile robot. Unless the environment includes something that acts like a corridor for the robot, it does not matter what motor output the robot generates, it cannot display the behaviour. The same goes for driving a car down a road (blind-folded or otherwise): without something equivalent to a car or a road, the behaviour is simply not possible.

For any particular behaviour the environment must contain a base set of environmental features that the agent can interact with in appropriate ways. Unless such features exist, the question of whether the agent produces the correct motor output to perform that behaviour is meaningless. In the swimming example, there must be a base set of features that the agent can interact with *as if it were* immersed in a liquid. In the corridor example, there must be a base set of features that the agent can interact with *as if it were* moving about in a corridor, and in the driving example there must be a base set of features that interact with the sensors and motors of the agent *as if it were* driving a car down a road. The point is that whether or not the agent *actually is* immersed in a liquid, or moving about in a corridor, or driving a car down a road is immaterial. To decide

[2]There are certain behaviours to do with the internal activity of the controller itself that this is not true of. Since there is no difference between simulation and reality for such behaviours, however, they shall not be considered here.

[3]Thanks to Joe Faith for this example.

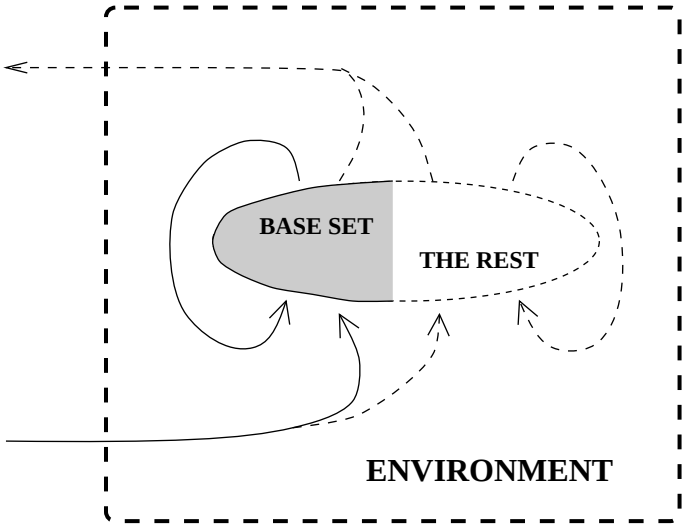

Figure 3.2: *This shows the analogue world of figure 3.1 divided up, with reference to a particular behaviour, into two dynamical systems: one that contains all the features of the behaviourally defined base set, and one that contains all those features of the world that can not affect the base set. The solid dark arrows represent the fact that the members of the base set must interact with each other and the output from the controller in appropriate behaviour-defined ways if the environment is to support the behaviour.*

whether or not the environment is capable of supporting the particular behaviour, we just need to know that it contains a base set of features that the agent can interact with as if it were.

The base set

This concept is so important to the discussion that follows in this thesis that we hereby give it a special name. From now on, the set of environmental features that must interact with each other and the controller's output in the appropriate behaviour-defined ways will be referred to as the *base set* with respect to that behaviour. Whether or not a particular environment is capable of supporting a behaviour is a function of whether or not that environment contains a suitable base set.

For a particular behaviour, the environmental features that make up the base set are all those that can affect whether or not the controller actually performs the behaviour on any given test. Thus the base set will include all those environmental features upon which behavioural success or failure is predicated and all the other environmental features and processes that can affect them. The important thing to realize is that the base set constitutes a dynamical system in its own right, instantiated as part of the larger dynamical system that is the analogue world, but affected from the outside only by output signals from the controller.

This is shown diagrammatically in figure 3.2. The analogue world of figure 3.1 has been divided up, with reference to a particular behaviour, into two dynamical systems: one that contains the behaviourally defined base set, and one that contains all those features of the world that *cannot* affect the members of this set. The solid dark arrows represent the fact that the members of the base set must interact with each other and the output from the controller in appropriate behaviour-defined ways if the environment is to support the behaviour.

The controller must produce the correct output

Given that the environment contains a suitable base set, whether or not a controller actually performs a particular behaviour depends on the output it actually produces, and specifically how this affects the way in which those features of the world that make up the base set change over time. A controller is said to *reliably* perform a particular behaviour if the way in which the members of the base set interact with each other and react to the controller's output constitutes an instance of the behaviour on *every* fitness trial. To sum up, a controller is said to have performed a particular behaviour over the course of a test period if two conditions are fulfilled:

1. The controller's environment must contain a suitable base set of features that may interact with each other and react to the controller's output in appropriate behaviour-defined ways.
2. The way in which the members of the base set *do* interact with each other and react to the controller's output over the course of the test period satisfies some suitable criterion for behavioural success.

In the next subsection, we use these conditions to state the circumstances under which a controller's behaviour in a simulation can be said to transfer into reality.

3.1.3 What counts as a successful transfer?

There are several different ways of judging whether a controller successfully transfers into reality after being evolved in a simulation, and the authors that have written about it so far use different methods. Miglino, Lund, and Nolfi (1995), for instance, look at the fitnesses of controllers in simulation and compare them to the fitnesses of controllers when downloaded into reality: the nearer the fitnesses the better the transfer. Jakobi, Husbands, and Harvey (1995), on the other hand, the authors use a more subjective approach to judge whether controllers behave qualitatively similarly in reality to how they behave in simulation. Neither of these methods will be used here, the main reason being that they are too general: we are not interested in how *every* controller that evolves in a simulation transfers into reality but only those that perform behaviours we are interested in. If we build a simulation in order to evolve light-seeking behaviours, for example, then it is of no consequence if a controller that jiggles on the spot in the simulation jiggles on the spot in a different fashion when downloaded into reality. What we will be interested in are how controllers that seek light in the simulation transfer into reality, and specifically whether they guide the real robot towards the real light.

As we shall see in section 3.3, the sort of minimal simulations that this thesis advocates building are constructed specifically for the evolution of particular robot behaviours. With this in mind, a very simple and straightforward criterion for successful transfer is adopted. Controllers are said to successfully cross the reality gap if, having evolved to reliably perform the particular behaviour we are after in simulation (according to the definition of reliable behaviour given above), they reliably perform the behaviour when downloaded into reality. It should be stressed that this does not necessarily involve any direct comparison between how the controller performs in simulation and how it performs in reality, but just a judgement of whether a controller that has evolved to reliably perform the particular behaviour in simulation, continues to reliably perform the behaviour when downloaded into reality. As in (Jakobi, Husbands, and Harvey 1995) this may often be a somewhat subjective measure, but should nevertheless be unambiguous in practice. If I use a simulation

to evolve controllers that cause a virtual robot to move around a cluttered environment avoiding objects, for example, then controllers successfully cross the reality gap if, when downloaded, they cause a real robot to move around a cluttered environment avoiding objects in a reliable fashion. If I use a simulation to evolve visually guided controllers that steer a virtual robot towards a target, then controllers successfully cross the reality gap if, when downloaded, they reliably steer a real robot towards the target in a satisfactory and acceptable manner.

3.2 Overcoming the failings of simulation

As has been demonstrated in several papers (Jakobi, Husbands, and Harvey 1995; Beer and Gallagher 1992; Nolfi, Miglino, and Parisi 1994) it *is* possible to evolve controllers in simulation for a real robot. However, the explanations offered by the authors of these papers as to why behaviours successfully transfer to reality when evolved under certain simulation conditions while not under others fall well short of the level of understanding necessary for the development of a general simulation building methodology. The consensus view seems to be that controllers will successfully transfer if the right amount of noise is included in a carefully constructed and empirically validated simulation of the robot and its environment[4]. But there is no such thing as the perfect simulation; some real-world features will be modelled at the expense of others. And since *my* empirically validated simulation might be *your* unrealistic toy-world we cannot proceed onto the question of what the perfect simulation for evolutionary robotics might look like without at least some agreement on how evolved controllers can and do transfer from simulations into reality in the first place.

If it were possible to build a perfect simulation of a real world robotics setup then crossing the reality gap would present no problems. This is because there would be no difference, from the point of view of the controller, between simulation and reality, and the behaviour it displayed in both would be identical. Unfortunately, any real-life simulation of a robot acting in the world will fail on two counts: it will only model a subset of the possible robot-environment interactions, and those that it does model, it will model inaccurately. Below, we examine each of these in turn and identify certain properties that evolving controllers must display if they are to overcome the failings of simulation and cross the reality gap. In this section, these properties are defined only in general terms to provide the reader with an initial understanding of the important concepts involved. As we shall see, they correspond respectively to the properties of being *being base set exclusive* and *base set robust*: both of which are more fully defined in section 3.3.

3.2.1 Simulations can't accurately model everything

Even the most comprehensive simulation can only hope to capture a subset of the totality of real world features and processes that make up a robot acting in the real world. The simulation built from this subset must nevertheless form a coherent whole from the point of view of evolving controllers, and the gaps left by the real-world features and processes that are *not* modelled must be filled in. However this is done, through filling in the gaps with arbitrary features or processes

[4] Although the nature of the 'right amount of noise', and indeed even what it means for a behaviour to 'successfully transfer from simulation to reality', varies markedly between papers on the topic

or whatever, this means that there will be aspects of even the most accurate of simulations which have no basis in reality.

For example, if a particular robot sensor blips high under certain circumstances and this real-world feature is not modelled, then the fact that the sensor *does not* blip high in the simulation under the same circumstances has no basis in reality. As another example, if the unknown probability distribution underlying the apparent stochasticity of a particular real world feature is arbitrarily modelled in the simulation by a normal distribution, then even if it has the same mean and standard deviation, there will be aspects of this normal distribution that have no basis in reality.

The problem this presents to controllers evolving in simulation is that they may come to rely on these aspects that have no basis in reality to perform their behaviour, and such controllers will more than likely fail to cross the reality gap. As long as controllers depend exclusively on those aspects of a simulation that *do* have a basis in reality, however, then this does not matter. This then is one of the conditions that must be met by evolved controllers if they are to overcome the failings of simulation. In order to cross the reality gap, the behaviour of evolved controllers must depend exclusively on those features and processes of the simulation that have been modelled on features and processes of the real world and on no other.

3.2.2 Simulations can't accurately model anything

Even if a controller evolves to depend exclusively on those aspects of a simulation that have a basis in reality, it may still not cross the reality gap. Modelling any set of real-world features and processes with 100% accuracy is just not possible, even with the most careful empirical validation; the environmental features and processes that a controller depends upon to perform its behaviour in simulation will inevitably differ from the real-world environmental features and processes they are modelled upon. To explain how controllers can and do cross the reality gap, therefore, requires an explanation of how a controller may continue to perform the same behaviour, even when there is a change to the environmental features and processes that this behaviour depends upon.

Consider first that, as explained in section 3.1.3, for a controller to cross the reality gap does not necessarily mean that it performs the task in exactly the same way in both simulation and reality. Thus small inaccuracies in the modelled features of a simulation may result in slight differences between a controller's behaviour in simulation and its behaviour in reality. As long as it continues to behave *satisfactorily* in reality then we may say that the controller has successfully crossed the reality gap. This is akin to hitting a barn door with a shot-gun at five paces. If your aim is off by a metre or so, you'll still hit it and this is all we are after. Of course in more complicated and involved situations there will not be so much room to manoeuvre, but it should be kept in mind that even for the most delicate of behaviours there will normally be a little bit of slop that can soak up small discrepancies.

Some controllers, however, will be more robust to inaccuracy than others. The level of model inaccuracy that an evolved controller can tolerate before it ceases to perform satisfactorily in the real world will depend on exactly *how* it uses the modelled features of the simulation to perform the task. For example, it has long been appreciated in the engineering world that processes which employ feedback or similar techniques will be far more robust to inaccuracy and noise than those that don't (Brogan 1991), and this is true here also; non-brittle control strategies and behaviours

that constantly correct themselves as they go, either through explicit feedback loops or implicitly via the environment (as in Braitenberg's vehicles (Braitenberg 1984)), lend themselves to the handling of the differences between simulation and reality. However, in certain situations even the most brittle, ballistic of control strategies will perform the task satisfactorily in reality, as in the shotgun example given above.

Unfortunately we cannot say anything much stronger than that controllers must be robust to the differences between the modelled features of a simulation and the real world features they are modelled on if they are to cross the reality gap. This is because there are so many ways of handling these differences depending on the behaviour, what we demand of it, the nature of the real-world features that are modelled in the simulation and so on. As we shall see, however, there may be ways of forcing the *evolution* of this type of robustness (in whatever form evolution cares to come up with), and there is therefore no need to focus too hard on exact mechanisms by which controllers may be robust since this job may be left to the evolutionary process itself. All that is important for our present purposes is to acknowledge that this type of robustness is a necessary property of controllers if they are to overcome the failings of simulation and successfully cross the reality gap

3.3 Minimal simulations

If the simulation approach to evolutionary robotics is to succeed then we must not only show how controllers can be evolved in simulation for real robots, we must also show how this can be done using simulations that run fast and are easy to design and build. Having defined what the reality gap consists of and given an account of the ways in which controllers can transfer across it, this section explores the concept of *minimal simulations*: the simplest type of simulation capable of evolving controllers for real robots. It starts by examining the minimal set of features that a simulation must include if the performance of a particular behaviour within that simulation is to be possible in the first place. It then goes on to examine the minimal relationships that these features must bear to reality if transfer across the reality gap is also to be possible. Having defined what a minimal simulation consists of, the section ends by restating, within the context of minimal simulations, the conditions that must be satisfied by a controller if it is to successfully transfer across the reality gap. In section 3.4, we go on to examine ways of evolving controllers that meet these conditions.

3.3.1 What features must be included?

In section 3.1.2 two conditions were given under which the motor output from a controller could be deemed to produce an instance of a particular behaviour. Of the two conditions, one was about the motor output itself but the other was to do with the environment. It stated that if the environment was to be capable of supporting the behaviour, then it must contain a suitable base set of environmental features that interact with each other and react to the controller's output in appropriate behaviour-defined ways. If the environment does not contain a suitable base set, it was claimed, then it is not even a meaningful question to ask whether a controller does or does not perform the behaviour in question. For this reason, logical necessity demands that any simulation for the evolution of a particular behaviour includes a suitable base set of features which interact

with each other and react to output signals from the controller in appropriate ways. Note that this is not the same as saying that the simulation contains a *model* of any particular base set of real-world features, but just that it contains a set of features that interact and react in appropriate behaviour-defined ways.

For most types of behaviour that we will be interested in, it is also necessary that evolving controllers receive a certain amount of input in some way from their environment. It is just not possible, for example, for a controller to evolve that can drive a simulated car down a simulated road in any sort of general way if that controller receives no input: the car will crash because the controller has no way of knowing where or when to steer. If it is necessary for certain features of the environment to affect controller input, furthermore, then it should be clear that these environmental features will be members of the base set. This is because the members of the base set are by definition the behaviourally relevant features of the environment; input originating from any other feature will therefore not be necessary for the successful performance of the behaviour. Again, this does not mean, from a logical point of view, that the simulation must necessarily model some particular real-world process by which controller input signals arise from some particular real-world feature, but just that the controller must receive *enough* input deriving from the relevant base set features of the simulated environment to make the performance of the behaviour logically possible within the simulation.

3.3.2 What relationships must these features bear to reality?

From a logical point of view, then, a simulation must include two things: a base set of environmental features that interact with each other and react to controller output in appropriate ways, and a sufficient number of processes by which the members of this base set can affect controller input. This, however, says nothing about the relationships that these features and processes must bear to the real world if transfer across the reality gap is to be possible - or just how minimal these relationships can be.

The first thing to realize is that we are only interested in whether controllers that reliably perform the behaviour in simulation continue to perform it in reality. We are not interested in how controllers that do not perform the behaviour in simulation behave when they are downloaded into reality. This has profound consequences for the amount of modelling that needs to be done, especially when one considers that the hardest aspects of some real-world situations to model are often those involved in the robot or robots *not* performing the behaviour rather than performing it. Thus, if we are evolving corridor following behaviour, the dynamics of the simulation might differ wildly from those of reality if the controller hits a wall or goes round in circles, but this does not matter since the controllers we are interested in transferring across the reality gap will neither hit walls nor go round in circles. Similarly, if we are evolving walking behaviour in an insect-like robot, the dynamics of the simulation might differ wildly from those of reality when the robot's legs clash or drag, but this will not matter since neither of these events occur during satisfactory and acceptable walking behaviour. The relationships that the logically necessary features of a simulation bear to reality may be arbitrary, therefore, when the behaviour is not being performed.

If crossing the reality gap is to be at all possible for controllers that *do* perform the behaviour, however, then they must be able to fulfill the conditions for successful transfer put forward in

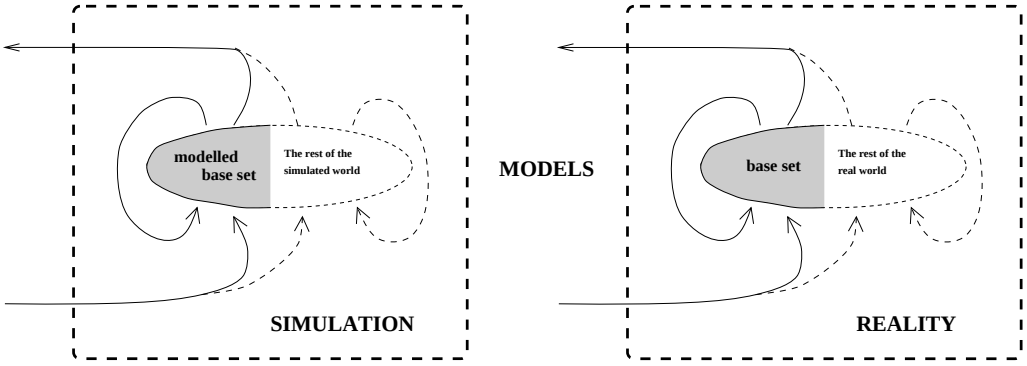

Figure 3.3: *This figure represents the relationship that a minimal simulation bears to reality. The features represented by the dark grey areas and solid black lines in simulation (the base set aspects) model the features represented by the dark grey areas and solid black lines in reality. The features represented by the white area and dashed lines in simulation (the implementation aspects) do not model the features represented by the white area and dashed lines in reality.*

section 3.2. The only demand this makes on the simulation (from section 3.2.1) is that it must contain a sufficient number of features and processes, that controllers may exclusively depend upon to perform the behaviour we are after, which have a sound basis in reality. By definition, the features and processes identified in the previous section are the logical minimum sufficient for successful behaviour. If these are all the minimal simulation consists of, furthermore, then it is these features and processes that successful controllers will depend upon to perform their behaviour. If crossing the reality gap is to be possible, therefore, then when (and only when) the behaviour is being performed, the logically necessary features and processes of the simulation must mirror the real world.

3.3.3 Overcoming the failings of minimal simulations

Figure 3.3 shows a diagram of the minimal amount of modelling that must be included in a simulation for the evolution of a specific robot behaviour. A base set of real-world features that are capable of underlying the behaviour must be identified, and the way in which these interact with each other and react to controller output signals when the behaviour is being performed must be modelled. In addition, a number of input processes by which the real-world base set gives rise to aspects of controller input signals when the behaviour is being performed must be identified and modelled. The number of these processes that are modelled must be sufficient to ensure that the controller receives enough information about its environment to make performing the behaviour possible.

Such a minimal simulation fails to be identical to the real-world situation it tries to model in the same two ways as any other simulation: it fails to model everything, and that which it does attempt to model it models inaccurately. The conditions under which controllers cross the reality gap given in section 3.2, therefore, are also true for minimal simulations. It is important for the discussion to come, however, to spell out exactly what these conditions consist of in a minimal simulation context. This is done below through the introduction of new terminology that will be used widely throughout the rest of this thesis.

Base set aspects and implementation aspects

As explained in section 3.2, the environmental features and processes of even the most comprehensive and empirically validated of simulations can be divided into two types: those that are inaccurately modelled on the real world, and those that have no counterpart in reality at all. The distinction between these two is an important one to keep clear, in the context of minimal simulations, and we hereby give it its own terminology. Since all the modelled features and processes of a minimal simulation will have something to do with the base set, they shall be referred to throughout the rest of the thesis as the *base set aspects* of the simulation. All those features and processes of a minimal simulation that have no counterpart in reality shall be referred to as the *implementation aspects* of the simulation.

In practice, the distinction between base set aspects and implementation aspects can be ambiguous. For instance, there will be aspects of the simulation that do not match up with reality due to modelling inaccuracy; should such aspects be base set aspects (because they are inaccurately modelled on the real world) or implementation aspects (because they have no basis in reality)? This is a matter of definition and is answered here as follows. If a minimal simulation is to be capable of evolving controllers that cross the reality gap then, as explained above, it must include a behaviourally defined base set of features and processes which are all modelled on the real world. Any aspects of these features and processes that do not match up with reality will be due to modelling inaccuracy, therefore, and it seems sensible to call them base set aspects. A minimal simulation must also include processes by which the modelled base set gives rise to controller input. However, it is only logically necessary to model enough (and not all) of these processes on the real world. There may therefore be aspects of controller input within the simulation which do not match up with reality and which are *not* due to modelling inaccuracy: they have no basis in reality at all. Aspects such as this can only be referred to as implementation aspects. For consistency's sake, any aspects of controller input that do not match up with reality due solely to modelling inaccuracy should again be referred to as base set aspects. Thus if an aspect of a minimal simulation does not match up with reality due to modelling inaccuracy then it is a base set aspect. If an aspect of a minimal simulation has no basis in reality and this is not due to modelling inaccuracy then it is an implementation aspect. The distinction between base set aspects and implementation aspects is shown diagrammatically in figure 3.3.

Base set exclusive

Reformulating the conclusions of section 3.2.1 using the terminology just introduced, a controller is only likely to transfer from a minimal simulation into reality if it depends exclusively on the base set aspects of the simulation, and those aspects alone, to perform the behaviour we are interested in. A controller is defined to be *base set exclusive* within a minimal simulation if it depends exclusively on the base set aspects in this way.

One important point to note here is that, according to the definition given above, the only way in which the implementation aspects of a minimal simulation can affect the controller's behaviour is through its input signals. This can be seen in figure 3.3 where the only way in which the white area marked 'the rest of the simulated world' can affect the grey area marked 'modelled base set' is via the dashed line leading off to the controller input. A controller is base set exclusive, therefore,

if it depends exclusively on the base set aspects of its input signals to perform the behaviour, and those aspects alone.

Base set robust

The base set aspects of a minimal simulation will always be slightly different to the real-world features and processes that they model. Even if controllers evolve to be completely base set exclusive, therefore, there will still be differences for them between the minimal simulation and the real world. In order to successfully transfer they must be robust to these differences. This is the condition on crossing the reality gap that was put forward in section 3.2.2. From now on, a controller that is robust to the differences between the base set aspects of a minimal simulation and the real world features and processes they model will be referred to as *base set robust*.

3.4 Evolving controllers to cross the reality gap

In the previous section we analysed just how minimal a simulation for the evolution of robot controllers can be and stated the properties that controllers which have evolved in such a minimal simulation must display if they are to successfully transfer into reality. Controllers that evolve to perform the behaviour we are after will not necessarily display these properties, however, and unless we possess techniques for forcing them to, we are no closer to a useful tool for the evolution of controllers for real robots than we were before. This section puts forward such techniques. It examines exactly how one might go about building simulations that are not only minimal in the sense put forward in the previous section, but which also force controllers that evolve to perform the behaviour we are after to be both base set exclusive and base set robust.

3.4.1 Evolving controllers to be base set exclusive

When evolving controllers in simulation to perform a specific task, a fitness criterion is used (usually an explicit function tailored to the task) to tell which controllers are fitter than others. If this fitness criterion is set up correctly, then all controllers that are able to consistently and robustly perform the task we are after will also be reliably fit and vice versa. If a single fitness evaluation consists of taking the average scores from several independent trials, then a reliably fit controller will score a high fitness value on all such trials. In such a situation, a controller that is base set exclusive is one that employs the base set aspects of its input, and these aspects alone, to be reliably fit.

One way of forcing this to happen is to make all the implementation aspects of a controller's input *unreliable* by varying them randomly from trial to trial. If this is done correctly, then the only practicable way in which a controller can be *reliably* fit, trial after trial, is by using those aspects of its input that are in themselves reliable i.e. the base set aspects. Since a single fitness evaluation involves several independent trials, reliably fit controllers will score more highly, in the long run, over those that are less reliable, and we may expect them to be selected for by the evolutionary process. If the process succeeds, and reliably fit controllers evolve, then we can be confident that they will rely exclusively on the base set aspects of their input to perform their behaviour, and will therefore be base set exclusive.

Before making the implementation aspects of a controller's input unreliable, they must first be identified. They will arise mostly as an incidental artifact of the modelling process, although they do not actually model anything themselves, which can make them quite subtle and hard to spot. For example, one of the base set aspects of controller input in a particular real-world situation might be that a particular sensor returns a value in the interval 0 to 13 when pointed in a certain direction. In implementing this interaction as a base set aspect of controller input in the simulation, however, we must chose a particular *way in which* values are returned between 0 and 13. Values could be returned from across the whole interval, or they could all equal 7. The point is that unless the way in which values are returned within this interval in simulation models the way in which they are returned in reality, then it is an implementation aspect of controller input and has no real-world basis. If the way in which values are returned within the interval 0 to 13 is randomly varied from trial to trial, however, then evolving controllers cannot rely on *how* they arise within this interval (the implementation aspect), but only on the fact that they do (the base set aspect).

In other cases, the implementation aspects of controller input signals will be obvious to us since we have put them in especially to make the modelling process easier or to reduce computational overheads. For example, we may note that the real-world base set aspects of controller input that we would like to include in our simulation are very simple to model when the robot is located within certain areas of its environment, and very hard to model in others. In order to make the job of building our simulation easier, therefore, we might include a model of the processes by which the real-world base set affect controller input which is accurate when the virtual robot is in the easy-to-model areas, but which is totally unrealistic when the robot is in the hard-to-model areas. In this case, the effects that these modelled processes have on controller input when the virtual robot is situated in the easy-to-model areas are base set aspects of the input, and the effects that they have on controller input when the virtual robot is situated in the hard-to-model areas are implementation aspects of the input. These implementation aspects can be made completely unreliable by randomly varying from trial to trial the effects that the modelled processes have on controller input when the virtual robot is situated in the hard-to-model area. Reliably fit controllers will employ strategies, therefore, that rely on the effects that the modelled processes have on controller input in the easy-to-model areas, while completely ignoring any effects that the modelled processes have on controller input when the robot is situated in the hard-to-model areas. Extra care must be taken in this sort of situation to ensure that the base set aspects of controller input within the simulation are comprehensive enough to allow reliably fit controllers to evolve. There is a real danger, if we are over-zealous in our lust for computational and modelling expediency, that we may effectively exclude so many real-world features from the simulation that what we are left with is insufficient for successful behaviour.

Once made explicit, we must then tackle the task of injecting randomness into the implementation aspects of controller input. In many cases it will be tempting to just add large amounts of noise to everything which is not a base set aspect and to leave it at that. However, if this noise is in itself reliable in the sense that evolving controllers can always count on it being there, then they can and will (Jakobi, Husbands, and Harvey 1995) evolve to use it to achieve high fitness. The secret is to randomly vary the implementation aspects of the controller input from trial to trial as opposed to just during each trial. Since a fitness evaluation consists of several trials, each

controller will be subjected to several different instances of each implementation aspect: noisy, absolute, black, white, big, small or whatever, depending on the nature of the particular aspect and the ways in which it can be varied. As long as there is nothing that all instances of a particular implementation aspect have in common, then reliably fit controllers will be totally independent of that aspect, or they will not be reliable.

Of course in practice it may be very difficult to ensure that there is nothing that all instances of any particular implementation aspect have in common. However, if the implementation aspects of the controller input within a simulation are made unreliable enough, then it is *so* much harder for evolution to find a way of using them reliably than it is for evolution to find a way of totally ignoring them that we can be extremely confident that controllers that evolve to be reliably fit will be base set exclusive.

3.4.2 Evolving controllers to be base set robust

In order to ensure that reliably fit controllers are base set robust, we must be able to ensure that they are able to cope with the differences between simulation and reality caused by inevitable modelling inaccuracy. We may approach this by adapting ideas borrowed from (Husbands and Harvey 1992) (with further elaborations in (Husbands, Harvey, and Cliff 1993b)). The idea is that by slightly varying, from trial to trail, every base set aspect of the simulation, reliably fit individuals will have to be able to cope with a certain amount of variation in order to be reliable. There will therefore be a selection pressure in favour of controllers that are better able to cope with slightly different base set aspects with slightly different dynamics, and thus in favour of controllers that are better able to cope with the differences between the base set aspects of the simulation and the real-world environmental variables, features and processes that they model. So in order to evolve reliably fit individuals that are base set robust, all we need is some way of knowing how much random variation it is necessary to apply to the base set features of the simulation and the best way in which to apply it.

As has already been said, it is rarely possible to simulate even the smallest portion of the world completely accurately. However, it is also rare that the simulation builder will not have at least some idea of how *inaccurate* their simulation is, and this seems a sensible way to work out limits on the amount of random variation we need to apply to the base set aspects of the simulation in order to evolve reliably fit controllers that are base set robust.

As to how this variation should be applied, there are lessons to be learnt from experiments reported in (Jakobi, Husbands, and Harvey 1995). In these experiments extra noise was added to the simulation over and above that present in reality and controllers were able to evolve that made use of the extra noise in such a way that they were reliably fit in simulation but failed miserably when down-loaded onto the real robot. However, this extra noise was *reliably* present during every trial, so evolving controllers that relied on its presence were still able to be reliably fit. In other words, evolving controllers were faced with the same base set aspects at every fitness trial, and these base set aspects were significantly different to the real-world features and processes that they modelled i.e. they were much more noisy. Given this fact, it is unsurprising that evolved controllers were unable to cross the reality gap.

In order for there to be a selection pressure in favour of controllers that can cope with slightly

differing versions of each base set aspect of a simulation, they need to be varied *between* trials, and not during them. There should of course be noise, during each trial, on base set aspects that model real-world entities such as sensors and actuators since there will also be noise on sensors and actuators in the real world. However, this noise should be regarded as an integral part of these base set aspects and not something extra to them. Noise levels should be altered between trials along with all the other base set aspects of a simulation. They should not be left steady throughout the evolutionary process at unrealistic levels.

3.5 How to build a minimal simulation for evolutionary robotics

In this section we put together all of the analysis and theory into a simple step-by-step recipe for building a minimal simulation for the evolution of a robot behaviour. The guide-lines put forward are very general and a fair amount of knowledge and insight is needed to tailor them to any particular real-world situation. However, it is hoped that together with the example simulations of chapters 5, 6, 7 and 8 the reader will be left with at least some idea of how they might go about constructing good minimal simulations for evolutionary robotics.

1. **Precisely define the behaviour.** Start by making a precise definition of the behaviour to be evolved. This should include both a description of the task to be performed by the robot(s) and the range of environmental conditions it is to be performed under.

2. **Identify the real-world base set.** Distinguish between those real-world features and processes that are relevant to the performance of the behaviour and those that are not. Those that are relevant constitute the base set. If possible, identify the way in which the members of the base set interact with each other and react to motor signals during the performance of the behaviour.

3. **Build a model of the way in which the members of the base set interact with each other and react to controller output (when the robot is performing the behaviour).** Make a model of the real-world base set of features and processes. The dynamics of this model need copy those of the real world only during the performance of successful behaviour. The dynamics when the behaviour is not being performed may often be shaped to smooth the fitness landscape, thus facilitating the evolution of successful controllers.

4. **Build a model of (enough of) the way in which the members of the base set affect controller input (when the robot is performing the behaviour).** Identify and model processes by which members of the real-world base set give rise to aspects of controller input. Just as with the model of the base set, these modelled processes need only copy their real-world counterparts when the behaviour is being performed. Make sure that the input aspects that these processes give rise to in the minimal simulation are sufficient to underly successful behaviour. Note that there are often several ways in which sufficient input processes may be identified and modelled, and the exact choice affects the possible strategies that evolving controllers may employ.

5. **Design a suitable fitness test.** Design a suitable fitness test that only awards maximum fitness points to those controllers that reliably perform the behaviour: some points to keep in mind are listed in section 2.5. In particular, each evaluation must involve a sufficient number of trials so that, with the right amount of trial to trial variation (see below), we can be confident that controllers which achieve high fitness in all of them are reliably fit, base set exclusive and base set robust.

6. **Ensure that evolving controllers are base set exclusive.** Make a distinction between the implementation aspects of controllers' input signals and the base set aspects. Those implementation aspects that are present during the performance of the behaviour must be randomly varied from

trial to trial so that evolving controllers that depend upon them are unreliable. In particular, enough variation must be included to ensure that evolving controllers can not, in practice, be reliably fit unless they are base set exclusive i.e. they actively ignore each implementation aspect and depend exclusively on the base set aspects of their input to perform the behaviour.

7. **Ensure that evolving controllers are base set robust.** Every base set aspect of the simulation must be randomly varied from trial to trial so that reliably fit controllers are forced to be base set robust. The extent of this random variation must be large enough that controllers which evolve to be reliably fit are also able to cope with the inevitable differences between the base set aspects of the simulation and their real-world counterparts. Care should be taken that this variation is not so large that reliably fit controllers fail to evolve at all.

3.6 The pros and cons of minimal simulations

Various different simulations have been used in the past to evolve controllers for robots: in (Jakobi, Husbands, and Harvey 1995), an empirically verified model of the underlying physics was constructed; in (Miglino, Lund, and Nolfi 1995), look-up-tables compiled from real-world sensor data were used; and in (Yamanuchi and Beer 1994), the factory-built simulation came supplied with the robot. However, the fundamental approach underlying the use of all of these simulations to evolve controllers for robots is the same: the less differences there are between simulation and reality, the less likely it is that controllers will evolve to depend on them and fail to download into reality. Such an approach is committed to building simulations that are as faithful and accurate a model of the real world as possible. It will not scale up.

The approach put forward in this chapter is very different. Instead of trying to eliminate the differences between simulation and reality they are acknowledged, and mechanisms are put in place to prevent evolving controllers from relying on them. Thus controllers are not only evolved in a minimal simulation to perform a specific real-world behaviour, they are *also* evolved to be robust to the differences between the minimal simulation and reality. A careful inspection of the arguments put forward in this chapter will reveal that nowhere is it implied that the base set aspects of a simulation should reflect reality as closely as possible, nor that the number of implementation aspects of a simulation should be kept to a bear minimum, and this is where the potential power of minimal simulations lies. A reliably fit controller that evolves in a simulation containing very inaccurate base set aspects and lots of implementation aspects is just as likely as any other to cross the reality gap *provided* that the right amount of random variation is included in the simulation in the right way according to the methodology laid out above.

What *is* much more likely in this situation, is that reliably fit controllers will fail to evolve at all. There will always be limits to the amount of randomness that the evolutionary machinery can find ways of coping with, no matter how this machinery is set up. If the amount of variation necessary to ensure that reliably fit controllers cross the reality gap surpasses this limit, then reliably fit controllers will just fail to evolve. However, if the evolutionary machinery *is* sufficiently powerful we can evolve complex controllers, capable of performing non-trivial real-world tasks, using surprisingly inaccurate and simple simulations. This means that how one chooses to model features of the world within a minimal simulation may be governed by considerations of computational expense or ease of construction rather than those of fidelity.

Minimalism has trade offs. By providing only a minimally sufficient base set of robot-environment

interactions within a minimal simulation, we are depriving evolution of its opportunistic ability, when performed within the real world, to ground reliable behaviour in any aspect of the environment it sees fit, whether we have thought of it or not[5]. However, this point also has its positive side. By making explicit and modelling only the *behaviourally relevant* features of a particular robotics setup, we ensure that the only controllers that can evolve to be reliably fit within the simulation will be those that are able to perform the 'real' task in a general non-brittle way, without relying on non-behaviourally relevant features that are specific to the particular environment within which they evolved. Thus the famous neural network, developed by the U.S. military, that could tell pictures of landscapes containing tanks apart from pictures of landscapes that did not - due to the unfortunate fact that all the pictures containing tanks were taken in the morning while those that did not were taken in the afternoon - could not evolve in a minimal simulation.

Chapter 4 presents a formal treatment of the general theory and methodology presented above. Chapters 5, 6, 7 and 8 each provide a practical example of a minimal simulation and detail experiments in which controllers were evolved using these simulations and transferred successfully onto real robots.

[5] Adrian Thompson's work with evolvable hardware (Thompson 1996b) provides a beautiful example of this type of artificial evolution.

Chapter 4

Minimal Simulations II: A Formal Treatment

This chapter presents a formalism for reasoning about controllers performing behaviours in environments and details an attempt to *logically derive* a minimal set of conditions for successfully crossing the reality gap from the same set of assumptions as those made in the last chapter. This attempt was only partially successful in that the derived conditions correspond closely to those put forward in the last chapter but are problematic to apply in practice due to the almost unavoidable clash between the universal quantifications required by logic and the finite nature of the real world. The insights afforded by this undertaking contributed greatly to the arguments and discussions of the last chapter, however, and the fact that the derived conditions correspond closely to those put forward in the previous chapter provides good evidence for the logical soundness of the theory underlying the minimal simulation approach. Details of the derivation are included here for the same reason as a mathematician includes the derivation of a theorem in a mathematics paper - if there is only analytic evidence to support a claim then this evidence must be provided. The reader who prefers their evidence to be empirical rather than analytic, however, may want to skip this chapter and go straight to the practical examples of minimal simulations described in chapters 5, 6, 7 and 8.

The first two sections of this chapter formalize the conceptual analysis of section 3.1: in Section 4.1 a notation is introduced for describing the way in which the coupled dynamical system that is a controller and its environment changes over time, and in Section 4.2 this is used to characterise what it means for a controller to perform a behaviour in such a system. This logical framework is then used in Section 4.3 to derive a minimal set of conditions for guaranteeing that a controller which performs a behaviour in one controller-environment system will continue to perform it in another. These conditions, which are of course the conditions on behavioural transference, are then listed in section 4.4. Finally, section 4.5 discusses how these formally derived conditions fit in with the general theory and methodology presented in chapter 3.

4.1 State-space equations for a general controller-environment system

The distinction between controller and environment can be done in a number of ways (Smithers 1994; Beer 1995a; Smithers 1997). For the purposes of this thesis, when the controllers we are

talking about are instantiated by a piece of computer code, we draw the line around the controller entirely in the digital domain (see section 3.1.1). This should be kept in mind as the formalism is put forward below. The software that receives digital input signals (in the form of an input vector) from the sensors and sends digital output signals (in the form of an output vector) to the motors is treated as the controller, and everything else including the actual sensors, motors and the controller's embodied form is treated as the environment. When defined thus, both controller and environment constitute separate dynamic systems, each with their own state, that are linked through the controller's input and output vectors. This section introduces a formalism that can be used to capture these interactions mathematically and puts forward general equations for the way in which they change over time. Before we do this, however, some basic notation is introduced:

> Let G be a dynamical system. The state of G at time t will be referred to as the state vector $\mathbf{g_t}$. The set of possible initial states g_0 that G can start from at time $t = 0$ will be referred to as G_{init}. The function which describes the way in which the value of $\mathbf{g_{t+1}}$ depends on $\mathbf{g_t}$ and any external inputs or controls $\mathbf{i_t}$ will be referred to as G_{diff}. Thus the equations
>
> $$\mathbf{g_0} \in G_{init} \quad \text{and} \quad \mathbf{g_{t+1}} = G_{diff}(\mathbf{g_t}, \mathbf{i_t}) \tag{4.1}$$
>
> completely capture the behaviour of the dynamical system G from time $t = 0$.

The controller

The controller constitutes a dynamical system in its own right whose trajectory through state space can be perturbed by the environment by way of the controller's input signals $\mathbf{i_t}$. If we call this dynamical system C, then $\mathbf{c_t}$ is the state vector of the controller at time t, C_{init} is the set of possible initial states $\mathbf{c_0}$and C_{diff} is the function that describes the way in which $\mathbf{c_{t+1}}$ depends on $\mathbf{c_t}$ and $\mathbf{i_t}$. Thus the equations

$$\mathbf{c_0} \in C_{init} \quad \text{and} \quad \mathbf{c_{t+1}} = C_{diff}(\mathbf{c_t}, \mathbf{i_t}) \tag{4.2}$$

completely describe how the controller's state changes over time in response to input signals.

The environment

The environment is also regarded as a dynamical system in its own right whose trajectory through state space can be perturbed by the output signals of the controller $\mathbf{o_t}$. If we call this dynamical system E, then $\mathbf{e_t}$ is the state vector of the environment at time t, E_{init} is the set of possible initial states $\mathbf{e_0}$ and E_{diff} is the function that describes the way in which $\mathbf{e_{t+1}}$ depends on $\mathbf{e_t}$ and $\mathbf{o_t}$. Thus the equations

$$\mathbf{e_0} \in E_{init} \quad \text{and} \quad \mathbf{e_{t+1}} = E_{diff}(\mathbf{e_t}, \mathbf{o_t}) \tag{4.3}$$

completely describe how the environment's state changes over time in response to the controller's output signals.

Controller-environment interaction equations

In order to completely describe the way in which a particular controller-environment system changes over time, we need to introduce two new functions that describe the way in which the input signals $\mathbf{i_t}$ to the controller C are a function of the state of the environment E at time t, and the way in which the output signals $\mathbf{o_t}$ to the environment E are a function of the state of

the controller C at time t. We will call these functions E_{out} and C_{out} respectively. These functions effectively couple the dynamical system C to the dynamical system E resulting in a coupled controller-environment dynamical system. Such a system shall be referred to throughout the rest of the chapter using the notation C $\rightleftarrows$ E. Thus the equations that completely describe how the dynamical system C $\rightleftarrows$ E changes over time are

$$\begin{array}{cc} \mathbf{c_0} \in C_{init} & \mathbf{e_0} \in E_{init} \\ \mathbf{o_t} = C_{out}(\mathbf{c_t}) & \mathbf{i_t} = E_{out}(\mathbf{e_t}) \\ \mathbf{c_{t+1}} = C_{diff}(\mathbf{c_t}, \mathbf{i_t}) & \mathbf{e_{t+1}} = E_{diff}(\mathbf{e_t}, \mathbf{o_t}) \end{array} \tag{4.4}$$

Substituting the second line into the third line we define a trajectory T_{CE} of C $\rightleftarrows$ E to be a sequence of pairs of states $< \mathbf{c}_i, \mathbf{e}_i >$ such that

$$\begin{array}{c} \mathbf{c_0} \in C_{init} \quad \mathbf{e_0} \in E_{init} \\ \mathbf{c_{t+1}} = C_{diff}(\mathbf{c_t}, E_{out}(\mathbf{e_t})) \\ \mathbf{e_{t+1}} = E_{diff}(\mathbf{e_t}, C_{out}(\mathbf{c_t})) \end{array} \tag{4.5}$$

In the next sections we will look at what it means for a controller to display a particular behaviour within such a system, and go on to ask what the conditions are under which a controller that displays a behaviour in one system will continue to display it in another.

4.2 What counts as an instance of a behaviour within a controller-environment system?

The conceptual analysis of section 3.2 concluded that a controller could only be deemed to perform a particular behaviour within a particular environment if:

1. The controller's environment contains a suitable base set of features that may interact with each other and react to the controller's output in appropriate behaviour-defined ways.
2. The way in which the members of the base set *do* interact with each other and react to the controller's output over the course of a test period satisfies some suitable criterion for behavioural success.

This section uses the formal notation developed so far in this chapter to express these two conditions as formal logic statements that must be true of a controller-environment system if the behaviour is to be performed.

If the ways in which the members of a base set of environmental features interact with each other and react to controller output are solely responsible for the future states of that base set, then the base set combined with the controller output forms a dynamical system of the same form as in (4.1). To require that an environment contains a suitable base set of features that may interact with each other and react to controller output in appropriate behaviour-defined ways (as in the first condition listed above), therefore, is the same as requiring that the environment contains a suitable base set of features that combine with controller output to form some appropriate behaviour-defined dynamical system. If for a particular behaviour β, say, we refer to the set $\{B^i\}$ of all such appropriate behaviour-defined dynamical systems as β_{env}, then an environment supports the particular behaviour β if and only if we can pick out some base set of features of the environment that may be combined with controller input to form one of the dynamical systems $B \in \beta_{env}$. Putting this more formally:

Let R_e be a reduction operator over state vector $\mathbf{e_t}$ of dynamical system E such that the vector $R_e(\mathbf{e_t})$ consists of an ordered subset of the variables that make up $\mathbf{e_t}$. Let $\beta_{env} = \{\mathrm{B^i}\}$ be the set of dynamical systems B that specifies all the different ways in which a subset of environmental features can interact with each other and react to controller output in order to make possible the performance of a particular behaviour. Environment E can support the behaviour if and only if

$$\begin{array}{l} \exists R_e \quad \text{and} \quad \exists \mathrm{B} \in \beta_{\mathrm{env}} \quad \text{s.t.} \\ \quad \mathbf{1}. \quad \forall \mathbf{e_0} \in E_{init} \quad R_e(\mathbf{e_0}) \in B_{init} \\ \quad \mathbf{2}. \quad R_e(\mathbf{e_t}) = \mathbf{b_t} \quad \Rightarrow \\ \qquad\qquad R_e(E_{diff}(\mathbf{e_t}, \mathbf{o_t})) = B^j_{diff}(\mathbf{b_t}, \mathbf{o_t}) \end{array} \tag{4.6}$$

If we *can* find a suitable R_e such that (4.6) is satisfied, then whether or not the controller actually *does* perform the behaviour within a given time interval is a function of how the variables that R_e picks out change over time. In other words, if $T_{CE} =< \mathbf{c}_0, \mathbf{e}_0 >, < \mathbf{c}_1, \mathbf{e}_1 > \ldots < \mathbf{c}_\tau, \mathbf{e}_\tau >$ is a trajectory of the system C $\rightleftarrows$ E, then whether this trajectory constitutes an instance of a particular behaviour β, say, will be a behaviour defined function of the sequence $R_e(\mathbf{e}_0), R_e(\mathbf{e}_1) \ldots, R_e(\mathbf{e}_\tau)$ obtained by applying R_e to every $\mathbf{e}_t$ in T_{CE} in turn. We will slightly bend the notation to refer to a sequence of $R_e(\mathbf{e}_t)$ obtained in this manner as $R_e(T_{CE})$. We define the behaviour-defined function β_{act}, which returns true or false, to be such that for any particular trajectory T_{CE},

$$\beta_{act}(R_e(T_{CE})) \Leftrightarrow \text{the controller exhibits behaviour } \beta$$

To sum up:

An controller is said to *reliably* perform the behaviour β in C $\rightleftarrows$ E if and only if

$$\begin{array}{l} \exists R_e \quad \text{s.t.} \\ \\ \mathbf{1}. \quad \exists B \in \beta_{\mathbf{env}} \quad s.t. \\ \quad (a) \quad \forall \mathbf{e_0} \in E_{init} \quad R_e(\mathbf{e_0}) \in B_{init} \\ \quad (b) \quad R_e(\mathbf{e_t}) = \mathbf{b_t} \quad \Rightarrow \\ \qquad\qquad R_e(E_{diff}(\mathbf{e_t}, \mathbf{o_t})) = B_{diff}(\mathbf{b_t}, \mathbf{o_t}) \\ \\ \mathbf{2}. \ \forall T_{CE} \quad \beta_{act}(R_e(T_{CE})) \end{array} \tag{4.7}$$

These two logic statements are the formal equivalents of the conditions for successful behaviour put forward in section 3.2.

4.3 When does a controller's behaviour in simulation imply its behaviour in reality?

In this section we formally derive a minimal set of conditions that must be true of a controller and a simulation if the controller's reliable behaviour in simulation is to guarantee its reliable behaviour in reality. The derivation precedes in three stages: from a completely hypothetical and idealized simulation to one that is realizable. This section is the most mathematical of the chapter and as such makes very little reference to real-world examples. In section 4.5 ways of turning theory into practice are discussed.

In the first stage of the derivation we consider a real world controller-environment system C $\rightleftarrows$ E and a simulation that consists of a single controller-environment system C $\rightleftarrows$ S. We show

that if S is identical to E, then any behaviour β that the controller reliably performs in simulation, it will also reliably perform in reality.

In the second stage of the derivation we again consider a real world controller-environment system C $\rightleftarrows$ E and a simulation that consists of a single controller-environment system C $\rightleftarrows$ S. In this scenario, however, all that E and S have in common is that they both support the same behaviour β, and in so doing, both 'track' the same dynamical system B $\in \beta_{\text{env}}$. We derive conditions on the simulation and controller under which the fact that the controller reliably performs β in simulation implies that it will reliably perform it in reality.

In the third stage we again consider a real world controller-environment system C $\rightleftarrows$ E, but our simulation consists of a whole set of different controller-environment systems $\{\text{C} \rightleftarrows \text{S}^{\text{i}}\}$. In this scenario, all S^{i} support behaviour β but only one $\text{S}^{\text{j}} \in \{\text{S}^{\text{i}}\}$ tracks the *same* dynamical system B $\in \beta_{\text{env}}$ as E. We extend the conditions derived in the previous stage under which the fact that the controller reliably performs β in simulation implies that it will reliably perform it in reality.

4.3.1 S = E

In this subsection we look at a hypothetical simulation consisting of a single controller-environment system C $\rightleftarrows$ S that is identical to a real-world controller-environment interaction system C $\rightleftarrows$ E and show that any behaviour that the controller reliably performs in C $\rightleftarrows$ S it will also reliably perform in C $\rightleftarrows$ E.

From Section 4.1, we can write controller-environment interaction equations for two systems, $C \rightleftarrows E$ and $C \rightleftarrows S$. Note however that the controller dynamical systems C are the same for both systems, so to avoid confusion the symbols for the state of each controller $\mathbf{c_t}$ include superscript symbols corresponding to the names of their environment dynamical systems.

$$\begin{aligned} \mathbf{c_0^E} &\in C_{init} \quad \mathbf{e_0} \in E_{init} \\ \mathbf{c_{t+1}^E} &= C_{diff}(\mathbf{c_t^E}, E_{out}(\mathbf{e_t})) \\ \mathbf{e_{t+1}} &= E_{diff}(\mathbf{e_t}, C_{out}(\mathbf{c_t^E})) \end{aligned} \tag{4.8}$$

and

$$\begin{aligned} \mathbf{c_0^S} &\in C_{init} \quad \mathbf{s_0} \in S_{init} \\ \mathbf{c_{t+1}^S} &= C_{diff}(\mathbf{c_t^S}, S_{out}(\mathbf{s_t})) \\ \mathbf{s_{t+1}} &= S_{diff}(\mathbf{s_t}, C_{out}(\mathbf{c_t^S})) \end{aligned} \tag{4.9}$$

Now if S = E then it is evident that every trajectory T_{CE} of $C \rightleftarrows E$ is a possible trajectory T_{CS} of $C \rightleftarrows S$. So if something is true of all possible trajectories T_{CS} of $C \rightleftarrows S$, it is also true of all possible trajectories T_{CE} of $C \rightleftarrows E$. Therefore

$$\beta_{act}(R(T_{CS}))\,\forall R(T_{CS}) \;\Rightarrow\; \beta_{act}(R(T_{CE}))\,\forall R(T_{CE})$$

4.3.2 S **and** E **track the same** B $\in \beta_{env}$

In this scenario, our simulation again consists of a single controller-environment system C $\rightleftarrows$ S and a real-world controller-environment interaction system C $\rightleftarrows$ E. All that S and E have in common, however, is that we can define reduction operators R_s and R_e for both systems such that for some

dynamical system B $\in \beta_{env}$,

$$\begin{array}{ll} \mathbf{1}. & \forall \mathbf{e_0} \in E_{init} \quad R_e(\mathbf{e_0}) \in B_{init} \\ \mathbf{2}. & R_e(\mathbf{e_t}) = \mathbf{b_t} \quad \Rightarrow \\ & \qquad R_e(E_{diff}(\mathbf{e_t}, \mathbf{o_t})) = B_{diff}(\mathbf{b_t}, \mathbf{o_t}) \end{array} \tag{4.10}$$

and

$$\begin{array}{ll} \mathbf{1}. & \forall \mathbf{s_0} \in S_{init} \quad R_s(\mathbf{s_0}) \in B_{init} \\ \mathbf{2}. & R_s(\mathbf{s_t}) = \mathbf{b_t} \quad \Rightarrow \\ & \qquad R_s(S_{diff}(\mathbf{s_t}, \mathbf{o_t})) = B_{diff}(\mathbf{b_t}, \mathbf{o_t}) \end{array} \tag{4.11}$$

This is a much looser constraint on the two controller-environment system than identity. In particular it means that the simulation does not need to model the whole of E, which is the entire universe when taken to its logical conclusion, but only the behaviourally relevant features. We shall now derive two conditions on the simulation and controller that must be fulfilled if the fact that the controller reliably performs β in simulation is to imply that it reliably performs β in reality.

Applying the relevant reduction operators to (4.8) and (4.9) we get

$$\begin{array}{c} \mathbf{c_0^E} \in C_{init} \quad \mathbf{e_0} \in E_{init} \\ \mathbf{c_{t+1}^E} = C_{diff}(\mathbf{c_t^E}, E_{out}(\mathbf{e_t})) \\ R_e(\mathbf{e_{t+1}}) = R_e(E_{diff}(\mathbf{e_t}, C_{out}(\mathbf{c_t^E}))) \end{array}$$

and

$$\begin{array}{c} \mathbf{c_0^S} \in C_{init} \quad \mathbf{s_0} \in S_{init} \\ \mathbf{c_{t+1}^S} = C_{diff}(\mathbf{c_t^S}, S_{out}(\mathbf{s_t})) \\ R_s(\mathbf{s_{t+1}}) = R_s(S_{diff}(\mathbf{s_t}, C_{out}(\mathbf{c_t^S}))) \end{array}$$

And we can deduce from (4.10) and (4.11) that

$$\begin{array}{l} R_e(\mathbf{e_t}) = R_s(\mathbf{s_t}) \\ \quad \Rightarrow \\ \qquad R_e(E_{diff}(\mathbf{e_t}, \mathbf{o_t})) \; = \\ \qquad\qquad\qquad R_s(S_{diff}(\mathbf{s_t}, \mathbf{o_t})) \qquad \forall \mathbf{o_t} \end{array}$$

therefore if we introduce the condition on C and S_{out} that

$$\begin{array}{l} R_e(\mathbf{e_t}) = R_s(\mathbf{s_t}) \\ \quad \Rightarrow \\ \qquad C_{diff}(\mathbf{c_t}, E_{out}(\mathbf{e_t})) \; = \\ \qquad\qquad\qquad C_{diff}(\mathbf{c_t}, S_{out}(\mathbf{s_t})) \qquad \forall \mathbf{c_t} \end{array} \tag{4.12}$$

then

$$\begin{array}{l} \mathbf{c_t^E} = \mathbf{c_t^S} \text{ and } \mathrm{R_e}(\mathbf{e_t}) = \mathrm{R_s}(\mathbf{s_t}) \\ \quad \Rightarrow \\ \qquad C_{diff}(\mathbf{c_t^E}, E_{out}(\mathbf{e_t})) \; = \\ \qquad\qquad\qquad C_{diff}(\mathbf{c_t^S}, S_{out}(\mathbf{s_t})) \\ \\ \qquad R_e(E_{diff}(\mathbf{e_t}, C_{out}(\mathbf{c_t^E}))) \; = \\ \qquad\qquad\qquad R_s(S_{diff}(\mathbf{s_t}, C_{out}(\mathbf{c_t^S}))) \end{array}$$

which is the same as saying that

$$\begin{array}{l} \mathbf{c_t^E} = \mathbf{c_t^S} \text{ and } \mathrm{R_e}(\mathbf{e_t}) = \mathrm{R_s}(\mathbf{s_t}) \\ \quad \Rightarrow \\ \qquad \mathbf{c_{t+1}^E} \; = \; \mathbf{c_t^S} \\ \qquad R_e(\mathbf{e_{t+1}}) \; = \; R_s(\mathbf{s_{t+1}}) \end{array}$$

and if we introduce a further condition on S that

$$\begin{array}{l} \forall \mathbf{e_0} \in E_{init} \quad \exists \mathbf{s_0} \in S_{init} \\ \quad s.t. \quad R_e(\mathbf{e_0}) = R_s(\mathbf{s_0}) \end{array} \tag{4.13}$$

then we can say by induction that

$$\begin{array}{c} \forall \mathbf{e_0} \in E_{init} \quad \exists \mathbf{s_0} \in S_{init} \\ s.t.\ R_e(\mathbf{e_0}) = R_s(\mathbf{s_0}) \\ \text{and} \\ R_e(\mathbf{e_t}) = R_s(\mathbf{s_t}) \quad \forall t \end{array}$$

In words, this means that provided condition (4.12) and (4.13) are true then every trajectory $R(T_{CE})$ of C $\rightleftarrows$ E is a possible trajectory $R(T_{CS})$ of C $\rightleftarrows$ S. Therefore if $\beta_{act}(R(T_{CS}))$ for all possible $R(T_{CS})$ of C $\rightleftarrows$ S then $\beta_{act}(R(T_{CE}))$ for all possible $R(T_{CE})$ of C $\rightleftarrows$ E.

At first sight, condition (4.12) seems contrived and unlikely. However, it is actually the same conditions as that first described in section 3.2.1. In words, it is the condition that all aspects of the input signals that can affect the controller's internal state must derive exclusively, and in the same way for both systems, from the base sets of environmental features that are picked out by R_e and R_s. In keeping with the terminology introduced in section 3.3.3, we shall refer to controllers that fulfill this condition within an appropriate simulation as being *base set exclusive*.

Condition (4.13) is much easier to reach an intuitive understanding of: for every possible start state in reality, there must be a possible start state in simulation which is identical with respect to the base sets of environmental features that are picked out by R_e and R_s.

4.3.3 $S^j \in \{S^i\}$ and E track the same $B \in \beta_{env}$

In practice, a simulation rarely models even a small portion of the real world 100% accurately. The situation in which S and E track the same dynamical system is, therefore, for the most part hypothetical. However, even if it is impossible to specify exactly a single S that tracks the same dynamical system as the real world environment E, a simulation builder may be able to define *a set* of dynamical systems $\{S^i\}$ (possibly by specifying parameter ranges rather than values) that *contains* an $S^j \in \{S^i\}$ that tracks the same dynamical system as the real world environment E. If the controller then reliably performs behaviour β in *all* C $\rightleftarrows$ S^j, and condition (4.12) and (4.13) of section 4.3.2 are fulfilled, then the controller will reliably perform behaviour β in reality.

More formally,

If the set $\{S^i\}$ is such that

$$\begin{array}{l} \exists S^j \in \{S^i\}\ s.t. \\ \\ \mathbf{1}.\ R_e(\mathbf{e_t}) = R_{sj}(\mathbf{s_t^j}) \\ \quad \Rightarrow \\ \quad\quad R_e(E_{diff}(\mathbf{e_t}, \mathbf{o_t})) \ = \\ \quad\quad\quad\quad R_{sj}(S^j_{diff}(\mathbf{s_t^j}, \mathbf{o_t})) \qquad \forall \mathbf{o_t} \\ \\ \mathbf{2}.\ \forall \mathbf{e_0} \in E_{init} \quad \exists \mathbf{s_0^j} \in S^j_{init} \\ \quad s.t. \quad R_e(\mathbf{e_0}) = R_{sj}(\mathbf{s_0^j}) \end{array} \tag{4.14}$$

and if the controller is base set exclusive for all $\{S^i\}$ such that

$$\begin{array}{l} \forall S^j \in \{S^i\} \\ \quad R_e(\mathbf{e_t}) = R_{sj}(\mathbf{s_t^j}) \\ \quad\quad \Rightarrow \\ \quad\quad\quad C_{diff}(\mathbf{c_t}, E_{out}(\mathbf{e_t})) = \\ \quad\quad\quad\quad\quad C_{diff}(\mathbf{c_t}, S^j_{out}(\mathbf{s_t^j})) \qquad \forall \mathbf{c_t} \end{array} \tag{4.15}$$

then the proof of section 4.3.2 goes through: every trajectory $R(T_{CE})$ of C $\rightleftarrows$ E is a possible trajectory $R(T_{SjC})$ of C $\rightleftarrows$ S^j.

Therefore since for all possible $S^j \in \{S^i\}$, $\beta_{act}(R(T_{SjC}))$ for all possible $R(T_{SjC})$, we know that $\beta_{act}(R(T_{CE}))$ for all possible $R(T_{CE})$ of C $\rightleftarrows$ E.

Note that this requires that

$$\forall S^j \in \{S^i\} \quad \forall T_{SjC} \quad \beta_{act}(R_{sj}(T_{SjC})) \tag{4.16}$$

which is a stronger condition on the controller's behaviour than that it just reliably performs behaviour β in one single controller-environment system C $\rightleftarrows$ S. If condition (4.16) is fulfilled then the controller must be able to perform behaviour β robustly with respect to variations in the way in which the behaviourally relevant base set of environmental features changes over time and responds to output signals. This is the same condition as that first described in section 3.2.2. In keeping with the terminology introduced in section 3.3.3, we say that a controller that fulfills condition (4.16) is *base set robust*.

4.4 Minimal conditions for behavioural transfer

We are now in a position where we can sum up the last section and put forward a minimal set of formal conditions on both controller and simulation that are minimally sufficient to ensure the controller's reliable behaviour in reality. The conditions have been roughly divided in half, two for the controller C and two for the set of simulation environments $\{S^i\}$.

Let β be a behaviour that defines a set β_{env} of dynamical systems B and a function β_{act} that returns true or false when applied to a sequence of the form $R_e(\mathbf{e}_0), R_e(\mathbf{e}_1) \ldots, R_e(\mathbf{e}_\tau)$.

Let C $\rightleftarrows$ E be a real-world controller-environment interaction system that supports behaviour β such that (4.6) is satisfied.

Let $\{C \rightleftarrows S^i\}$ be a set of controller-environment interaction systems that together constitute a simulation, and each of which supports behaviour β such that (4.6) is satisfied.

Now if

$$\exists S^j \in \{S^i\} \; s.t.$$

$$\begin{array}{l} \mathbf{1}.\ \forall \mathbf{e_0} \in E_{init} \quad \exists \mathbf{s_0^j} \in S^j_{init} \\ \quad s.t. \quad R_e(\mathbf{e_0}) = R_{sj}(\mathbf{s_0^j}) \end{array}$$

$$\begin{array}{l} \mathbf{2}.\ \forall < \mathbf{c_t^{s^j}}, \mathbf{s_t^j} > \in T_{CSj} \; s.t. \; \beta_{act}(R_{sj}(T_{CSj})) \\ \quad \mathbf{c_t^E} = \mathbf{c_t^{s^j}} \text{ and } \mathrm{R_e}(\mathbf{e_t}) = \mathrm{R_{sj}}(\mathbf{s_t^j}) \\ \quad\quad \Rightarrow \\ \quad\quad R_e(E_{diff}(\mathbf{e_t}, C_{out}(\mathbf{c_t^E}))) = \\ \quad\quad\quad\quad R_{sj}(S^j_{diff}(\mathbf{s_t^j}, C_{out}(\mathbf{c_t^{s^j}}))) \end{array} \tag{4.17}$$

and

$$\forall S^j \in \{S^i\}$$

$$\textbf{1}.\ \forall T_{CSj} \quad \beta_{act}(R_{sj}(T_{CSj}))$$

$$\begin{aligned}\textbf{2}.\ \forall < \mathbf{c_t}, \mathbf{s_t^j} > \in T_{CSj}\ s.t.\ \beta_{act}(R_{sj}(T_{CSj})) \\ R_e(\mathbf{e_t}) = R_{sj}(\mathbf{s_t^j}) \\ \Rightarrow \\ C_{diff}(\mathbf{c_t}, E_{out}(\mathbf{e_t}))\ = \\ C_{diff}(\mathbf{c_t}, S^j_{out}(\mathbf{s_t^j}))\end{aligned} \tag{4.18}$$

then

$$\forall T_{CE} \quad \beta_{act}(R_E(T_{CE}))$$

Conditions (4.17.1) and (4.18.1) are versions of conditions (4.14.2) and (4.16) respectively. Note however that conditions (4.17.2) and (4.18.2) are scoped differently to conditions (4.14.1) and (4.15) from which they are respectively derived. The qualification

$$\forall < \mathbf{c_t^{s^j}}, \mathbf{s_t^j} > \in T_{CSj}\ s.t.\ \beta_{act}(R_{sj}(T_{CSj}))$$

means that these conditions are only required to hold true for those trajectories of the simulation that constitute instances of the controller actually performing behaviour β. But, as explained less formally in section 3.3.2, this is all we need. If condition (4.18.1) is true, then this will be the only sort of trajectory that the simulation is capable of anyway.

4.5 Turning theory into practice

This section compares how these logically derived conditions on crossing the reality gap fit in with the practical methodology for building minimal simulations put forward in chapter 3. Specifically it discusses whether minimal simulations built according to this methodology fulfill conditions 4.17.1 and 4.17.2 above, and also whether controllers evolved in such minimal simulation fulfill conditions 4.18.1 and 4.18.2 above. Each is examined in turn.

In order to fulfill conditions 4.17.1 and 4.17.2 a simulation must model not one but a set of environments. At least one of these environments, furthermore, must contain a base set of environmental variables that interact with each other and react to controller output during successful behaviour in *exactly* the same way as the real-world base set. In practice, however, these conditions can rarely be completely fulfilled. Minimal Simulations do indeed model a set of environments whose dynamics are designed to 'contain' that of the real thing due to the way in which the base set aspects of the simulation must be varied from trial to trial (see section 3.4.2). However, most modelling of a real-world dynamical system will involve a simplification, and this is a difference which no amount of parameter-twiddling will render more complex. It is therefore extremely unlikely that the dynamics of any single member of this set will match those of the real world. Nevertheless, even if the simulation involves a simplified model, if a controller performs the desired behaviour in every environment S^j instantiated by this model (i.e. fulfills condition 4.18.1), then it *is* robust to small changes in the underlying dynamics of its world. If the set $\{S^i\}$ is large and varied enough, therefore, it is extremely unlikely that the mechanisms employed by reliably fit controllers to cope with all the various $\{C \rightleftarrows S^i\}$ will not also be sufficient to cope with the

differences between the closest $C \rightleftarrows S^j$ and $C \rightleftarrows E$ (for a discussion of what these mechanisms might look like see section 3.2.1).

It should also be noted that condition 4.17.2 corresponds to the arguments put forwards in section 3.3.2: we do not need to build the simulation so that some $S^j \in \{S^i\}$ tracks the same $B \in \beta_{env}$ as E for *all* trajectories of the two systems. The dynamics need only be the same or similar for those states of the simulation that occur when the controller is actually performing behaviour β.

In order to fulfill condition 4.18.1, controllers must be able to reliably perform the behaviour in question in all of the environments that can be instantiated by the simulation. In order to be reliably fit (from the definition in section 3.4.1), a controller must achieve high fitness on all of the trials that make up a fitness evaluation. Thus controllers that evolve to be reliably fit fulfill something close to condition 4.18.1 since each trial instantiates a different environment (see section 3.4.2). The difference is that a controller may only be tested on a finite number of environments in a single fitness evaluation, and we can not be sure therefore that it will be able to reliably perform the behaviour in question in *all* of the environments that can be instantiated by the minimal simulation. However, if a fitness evaluation contains a sufficient number of trials, then a controller which reliably scores high fitness on all such trials *is* robust to change in the base set aspects of its environment, even if we cannot be sure that it is 100% robust to all types of change. Furthermore, in order to propagate through a population of evolving individuals, and persist generation after generation within that population, evolving controllers (and their descendents) must score highly in many complete fitness evaluations. We may therefore be extremely confident that controllers which evolve to be reliably fit within a minimal simulation will be robust enough to variation in the base set aspects that it will also be robust to the variation between these aspects and their real-world counterparts.

In order for a controller to fulfill condition 4.18.2, two things need to be true: at least some of the processes by which the real-world base set gives rise to aspects of the controller‘s input must be modelled 100% accurately in the simulation, and the controller’s behaviour must depend exclusively on these aspects and these aspects alone. In a minimal simulation, of course, such processes can not be modelled 100% accurately, and the fact that this is not taken into account by condition 4.18.2 is a shortcoming of the formalism; in a minimal simulation, the modelled processes by which the members of the base set affect controller input are treated like any other base set aspects of the simulation and varied from trial to trial (see section 3.3.3). The second part of this condition corresponds precisely to the condition of being base set exclusive described in section 3.3.3. Since the implementation aspects of controller input are randomly varied from trial to trial in a minimal simulation in a way that makes them completely un-reliable (as prescribed by section 3.4.1), reliably fit controllers are forced to be base set exclusive and thus, as far as possible, to fulfill condition 4.18.2.

Ideally, the conditions of section 4.4 could be applied in practice. This would involve identifying those features of a particular controller, simulation and real-world setup that correspond to the terms of equations 4.17.1, 4.17.2, 4.18.1 and 4.18.2, and testing empirically to see if the relationships between them correspond to the relationships described by the equations. The less formal conditions of chapter 3 could therefore be thrown away and we would be left with precise

conditions whose satisfaction would *logically guarantee* the successful transferrance of controllers from simulation into reality.

Unfortunately this can not be done in practice for two reasons:

- Many of the terms of equations 4.17.1, 4.17.2, 4.18.1 and 4.18.2 are defined with reference to the dynamical system E which, in principle, describes the entire real world. In order to relate these terms to features of a particular real-world situation, however, we must first agree on a mapping between the variables of E and the features of the real world. Although such a mapping is possible 'in principle' it is not possible in practice since e.g. we would have to account for every feature of the entire real world to avoid performing an ad-hoc reduction before we mean to.
- The logic of the conditions of section 4.4 employs universal quantifiers and there are only finite resources at our disposal for evaluation purposes. For example in order to test that equation 4.18.1 is true, we would have to test that every possible trajectory of the controller within the simulation constituted an instance of the behaviour, and while this may not be an infinite number it is large enough to be impractical.

Thus we are left with the conclusion that the formalism can only be applied in principle and not in practice. Does this mean that it serves or has served no real purpose? Answer no:

- The conditions of section 4.4 correspond closely to those of chapter 3. The assumptions made in each case, and the way in which the arguments progress, can also be seen to correspond closely. This provides good evidence for the logical soundness of the arguments that led to the conditions for successful behaviour transferrance put forward in chapter 3.
- The development of the formalism, and the derivation of the conditions of section 4.4, necessitated a deep examination of the issues and led to insights into the problem which may not have arisen otherwise. There is no doubt that chapter 3 would have looked very different if the work put forward in this chapter had not been undertaken.

Thus the work presented in this chapter backs up and contributed to the less formal theory and argument of chapter 3, and for these reasons it is worth including. However, it cannot be applied directly in practice due to the reasons given above. This does not matter, provided we are prepared to admit that the general theory and methodology of chapter 3 does not *guarantee*, in any logically strong sense, that controllers which evolve to be reliably fit within a minimal simulation will transfer into reality. Careful thought as to the number of trials that make up a fitness evaluation and the extent to which the base set aspects are varied between trials makes it extremely unlikely that they won't transfer, however, and this is good enough for practical purposes. As empirical evidence of this, chapters 5, 6, 7 and 8 provide practical examples of minimal simulations and detail experiments in which controllers were evolved using these simulations and transferred successfully onto real robots.

Chapter 5

A minimal simulation of a Khepera robot

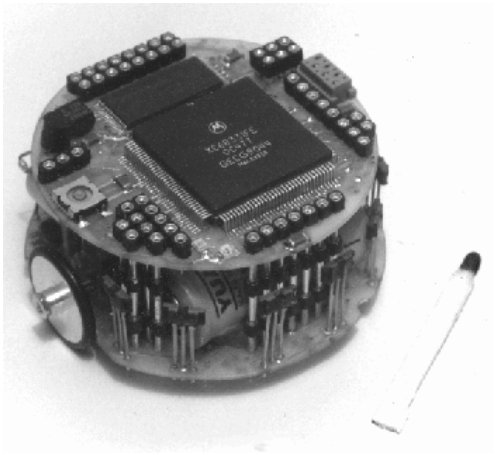

Figure 5.1: *The Khepera robot.*

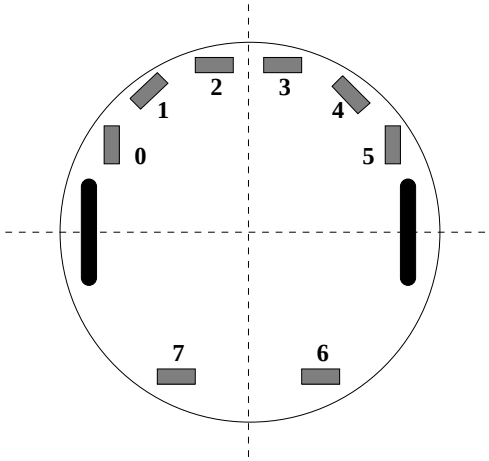

Figure 5.2: *A plan of the Khepera robot showing the positions and numbers of the infra red sensors and the two wheels.*

This chapter describes experiments in which neural network controllers were evolved for a Khepera robot using a minimal simulation. This robot, shown in figure 5.1, is 5.8 cm in diameter and about 3 cm high. It has eight infra red sensors, which respond to nearby objects, placed around the robot body as shown in figure 5.2. In a different mode these sensors may also be used to detect ambient light levels in the vicinity of the robot with very rough directional sensitivity (K-Team 1993).

Several different groups (Jakobi, Husbands, and Harvey 1995; Miglino, Lund, and Nolfi 1995; Michel 1995) have built Khepera simulators on which they have successfully evolved controllers that cross the reality gap. These simulators, however, were built to faithfully reproduce the real-world as accurately as possible, so seeing whether an inaccurate minimal simulation of a Khepera robot is equally capable of evolving controllers that cross the reality gap provides a good test of the theory and methodology of chapter 3.3. Specifically, experiments involving a minimal simulation of a Khepera that includes noise over and above that present in reality (as prescribed in section 3.4)

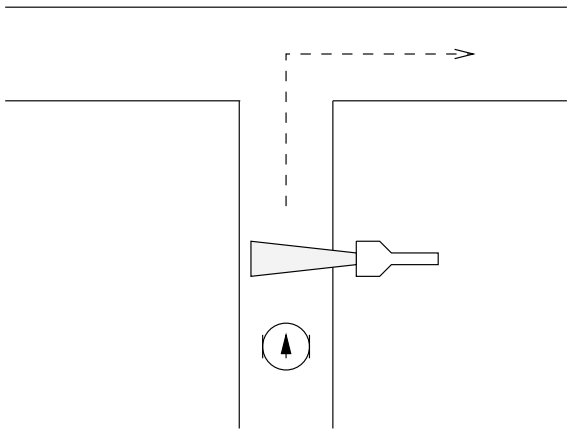

Figure 5.3: *The task in the real world.*

can be directly contrasted with the experiments reported in (Jakobi, Husbands, and Harvey 1995). Here controllers *failed* to download from a simulation of a Khepera robot into reality when too much (or too little) noise was present in the simulation. As argued in section 3.4, however, it was because this noise was reliably present that controllers could evolve to depend on it to perform their behaviours. The solution proposed in section 3.4 was to make each fitness evaluation the result of a number of individual fitness trials and to vary the amount and character of noise from trial to trial. This ensures that any noise is present in ways that controllers are extremely unlikely to evolve to rely on to perform their behaviours since the noise is itself unreliable. The experiments reported in this chapter represent a good test of this proposed solution.

The aim of the experiments was to evolve a behaviour for the Khepera robot that was at least one step up from the simple reactive behaviours that have been prevalent in the Evolutionary Robotics literature so far. The behaviour that was decided on is shown diagrammatically in figure 5.3. As a Khepera robot begins to negotiate a T-maze, it passes through a beam of light shining from one of the two sides, chosen at random. To score maximum fitness points the control architecture must 'remember' on which side of the corridor the light went on and, on reaching the junction, turn down the corresponding arm of the T-maze. This behaviour involves several elements: not only must controllers guide the robot down the corridors without touching the sides and negotiate the junction at the end of the first corridor (simple reactive behaviours both), but they must also involve some internal state that allows them to 'remember' which side the lamp was on so that they can take the correct turning at the junction.

5.1 The minimal simulation

The key observation upon which the minimal simulation was based was that, with respect to the infra-red sensors of a Khepera which have a maximum range of only about 8cm, a T-maze was identical to an infinite corridor almost everywhere. Where a T-maze differed from a corridor, at the T-junction, the interactions between the sensors and the corridor walls could be treated as implementation aspects of the simulation, and randomly varied from trial to trial according to the methodology laid out in section 4.15. Reliably fit controllers could therefore be forced to use

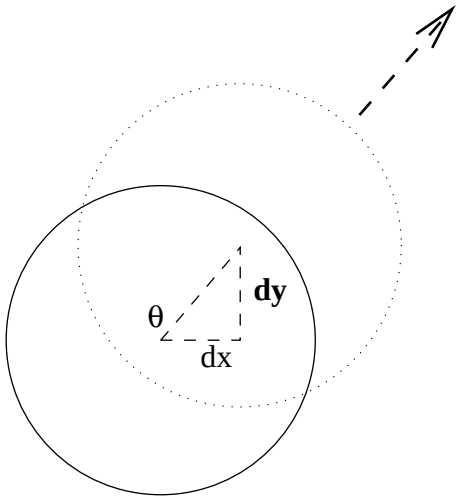

Figure 5.4: *A look up table holds horizontal and vertical increment values for 36 different orientation values and an average speed of 1.*

strategies that depended on the interactions between the infra-red sensors and the sections of the walls of the T-maze that could be regarded as straight and continuous corridor walls, and those interactions alone. In this way, the infra-red sensor model employed by the minimal simulation was constructed from two different phases of a simple continuous corridor model. Below, the step-by-step framework of section 3.5 is used to explain how the minimal simulation was constructed.

Precisely define the behaviour

As a Khepera robot begins to negotiate a T-maze, it passes through a beam of light shining from one of the two sides, chosen at random. To score maximum fitness points the control architecture must 'remember' on which side of the corridor the light went on and, on reaching the junction, turn down the corresponding arm of the T-maze.

Identify the real-world base set

Whether or not the controller performs the behaviour is a function of the robot's path through the corridor system in relation to the side the light was on. The features of the world that can affect this path are those that make up the causal pathway from controller output to movement of the robot within its environment. These include the way in which controller output affects the movement of the wheels, and the way in which wheel motion affects the position of the robot within the corridor system.

Build a model of the way in which the members of the base set interact with each other and react to controller output (when the robot is performing the behaviour).

The simulation was updated the equivalent of ten times a second. Figure 5.4 shows how the new position of the virtual Khepera within its environment was calculated at each iteration. The orientation was used as an index to a look-up-table with 36 pairs of values: horizontal and vertical increments for a Khepera travelling at a speed of 1cm per second. To work out its new position,

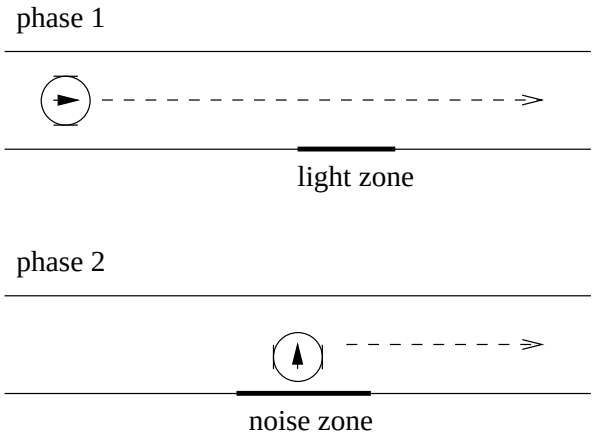

Figure 5.5: *Two phases of a simple infinite corridor model provide a rough simulation of how the infra red sensors respond in a T-maze environment.*

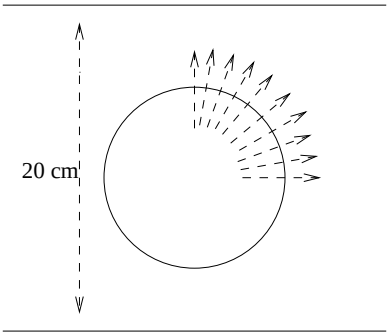

Figure 5.6: *A look up table contains the perpendicular distances to the walls of a 20cm wide corridor for all eight sensors in ten possible orientations.*

the values returned from this look up table were multiplied by the average wheel speed in cm per second. The speed of each wheel was calculated directly from multiplying the motor signals by the constant 0.8 cm per motor unit per second. The change in orientation at each iteration was equal to the difference between the distances the two wheels moved divided by the radius of the robot (about 5.2cm). There was no allowance for momentum and the noise inherent in the real-world situation was not modelled.

Build a model of (enough of) the way in which the members of the base set affect controller input (when the robot is performing the behaviour).

Figure 5.5 shows how the sensor values for the simulated robot were calculated in two separate phases in the simulation. In the first phase, when the robot was travelling down the first arm of the T-maze, the sensors were calculated as if the virtual robot was travelling down a simple infinite corridor. When the robot reached the junction of the T-maze the way in which the sensors were calculated suddenly changed for phase 2: as if the robot was suddenly popped out of the first corridor, rotated through ninety degrees, and popped into the middle of a second corridor.

Now although the way in which the simulated sensors returned values in this twin-corridor set-up varies significantly from the way in which the sensors return values in a real T-maze, the simulation had enough in common that evolving control architectures which were prohibited from relying on any of the differences were still able to sense enough of their environment to perform the task successfully. In particular, the robot-environment interactions governing the way in which the robot, travelling down a straight corridor, is confronted with a wall straight across its path and a second corridor stretching off to either side were all modelled.

Calculating the infra-red values as if the Khepera was within an infinite corridor proceeded in three stages. Firstly, the robot's orientation was used to generate rough distance-to-wall metrics for each sensor, as if the robot was in the centre of a 20cm wide infinite corridor. Secondly, these values were scaled according to the actual width of the corridor in the simulation, and the distance from the robot to each wall. And thirdly, the scaled distance-to-wall metrics were used to calculate

infra red sensor values by way of a simple linear relationship. This process is described in more detail below.

Figure 5.6 demonstrates what the values held in the infra-red look-up-table were based upon. There were 10 sets of 8 values, each set corresponding to one of 10 different robot orientations from facing straight down the corridor to perpendicularly facing one of the walls. The values themselves were based on the distances from the centre of the robot (which is 10cm away from each wall), along the lines of the corresponding sensors, to the walls of an infinitely long corridor. However, they were in fact always slightly shorter than the line of sight distance to the wall in order to account (in a very approximate way) for the fact that the infra-red sensors of a Khepera are sensitive over a whole arc rather than just along the direct line of sight extending out from each sensor (K-Team 1993). If the distance from the centre of the robot, along the line of a sensor, to a wall of the corridor was d, then the warped distance value wdv stored in the look-up-table was given by:

$$wdv = 10 + (d - 10)/3$$

The constant 3 in the denominator was decided upon fairly arbitrarily and without accurate measurement purely on the basis that it gives the equation roughly the right properties.

The minimum possible value stored in the table, therefore, for a sensor directly facing the wall, was 10cm. The maximum possible value, for a sensor directly facing down the corridor, was infinity. Now although there were only 10 sets of values stored in the table (one set of 8 for each multiple of 10 degrees between 0 and 90 degrees inclusive), it was a simple matter to calculate sets of values for any other multiple of 10 degrees. This was done by taking the particular orientation angle in question and rotating it by the appropriate multiple of 90 degrees until it lay in the correct quadrant. The look-up-table was then used to ascertain a set of values and these were reflected across the mid-line of the robot if necessary (i.e. if the angle was between 90 and 180 or between 270 and 360). If the robot was in the centre of a 20cm wide, infinitely long corridor, therefore, warped distance values could be calculated for any sensor at any orientation.

In practice, the perpendicular distance from the centre of the robot to a particular wall of the corridor was variable. Values were scaled appropriately, however, by multiplying all values returned from the look-up-table (for sensors that pointed at that particular wall) by the fraction attained by dividing the actual distance to the wall by 10cm. For example, if the distance from the robot to a wall was actually 5cm instead of 10cm, then look-up-table values for sensors that pointed at that wall were halved. In this way, the 80 values of the look-up-table were sufficient to find the approximate distance, warped according to the equation given above, from the centre of the robot in any position and any orientation, along the line of any sensor, to the wall of an infinite corridor of any width.

Having ascertained warped distance values wdv for each sensor, the actual value that each simulated sensor returned, V, was given by a simple linear function:

$$V = \begin{cases} 0 & wdv > a \\ 1024 \times (7 - wdv)/2 & a > wdv > b \\ 1024 & b > wdv \end{cases} \tag{5.1}$$

where a and b were the maximum and minimum extent, respectively, of the linear part of the response function. This meant that a sensor would saturate at maximum value if its warped distance value was less than b (randomly varied around an average of 5, see below on ensuring controllers are base set robust), would return zero if its warped distance value was greater than a (randomly varied around an average of 9, see below on ensuring controllers are base set robust), and would respond linearly in between.

A simple multiplicative congruential random number generator (Press, Vetterling, Teukolsky, and Flannery 1992) was used to generate uniformly distributed random deviates in the range ± 50. These were added to returned sensor values at each iteration. In addition, the lowest value an infra-red sensor could return was a random background value between 0 and 20. These noise levels roughly approximate the levels observed in the real world, and as such were as much a part of the robot-environment interaction model as any other aspect.

The way in which ambient light sensors respond to bright versus ambient light levels was modelled by a single line of code. When the robot entered a particular section of the corridor in phase 1 (that was randomly predefined in terms of length and position relative to the starting point), the values returned by the ambient light sensors on one side of the robot dropped from their normal background value of around 450 to a value of around 100, as if they had been illuminated by a bright light. When the robot left the special light zone the values returned to their background levels. Whether the right side of the robot or the left side was illuminated depended on which side of the corridor the light source was placed, and which of the two directions directly down the corridor the robot was closest to pointing. Random deviates in the range ± 50 were added to each ambient light sensor value at each iteration.

Design a suitable fitness test.

The fitness function returned the average value scored by an individual in a total of ten fitness trials, each lasting the equivalent of fifteen seconds. For half of these trials the light signal came from the right hand side of the corridor and for the other half it came from the left. At the end of each trial, the fitness value was calculated from the equation

$$T = \begin{cases} d_1 + d_2 + 100 & \textit{right way at lights} \\ d_1 + d_2 & \textit{wrong way at lights} \end{cases}$$

where d_1 was the distance that the virtual robot had travelled down the first corridor during the trial, d_2 was the distance it had travelled down the second corridor, and the bonus it got for going the right way at the T-junction was 100.

Ensure that evolving controllers are base set exclusive

The major differences between the simulation and the real world T-maze, as far as the sensors were concerned, all occurred around the T-junction. When the virtual Khepera suddenly appeared in the second corridor facing the wall at the start of phase 2, there was a continuous wall directly behind it (see figure 5.5). In reality, when the Khepera was confronted with a wall across its path and forced to make its decision on which way to turn, there was a complicated junction in the wall behind it (see figure 5.3). Because of this, the simulated robot's infra red sensor interactions with

the simulated back wall, in an area corresponding to where the corridors meet in reality and about 5cm to either side, were regarded as implementation aspects. If this section of the wall fell within range of the infra-red sensors then how these sensors reacted varied randomly from trial to trial - sometimes they returned maximum values, sometimes low values, and sometimes totally random values. In this way reliably fit controllers were forced to employ strategies that, at the decision point, were oblivious to this difference between the simulation and reality, relying only on the fact that there was a straight continuous wall in front of them and space to either side.

Ensure that evolving controllers are base set robust

To ensure that reliably fit controllers were base set robust many of the base set aspects of the simulation were varied slightly from trial to trial. The amount by which each parameter was varied was judged from a knowledge of the inaccuracy of the model to be large enough that controllers would evolve to be base set robust, but not so large that controllers would fail to evolve to be reliably fit in the first place. These included:

- The width of the two corridors: between 13cm and 23cm.
- The exact starting orientation of the robot: between ± 22.5 degrees of facing straight down the corridor.
- The length of the illuminated section of the corridor: between 2cm and 12cm.
- The total length of the corridor in phase 1: between 40cm and 60cm
- Random offsets of between $\pm$1cm per second were generated at the beginning of each trial, and added to wheel-speeds during position update calculations.
- The constants a and b of equation 5.1 were randomly set at the beginning of each trial, the former in the range 7.1 to 10.1 and the latter in the range 4.1 to 6.1.

In this way reliably fit controllers were forced to employ non-brittle strategies that could perform the behaviour in a whole range of different shaped T-maze environments and from a whole range of starting orientations. They were also forced to cope with robots with different motor and sensor characteristics.

5.2 The evolutionary machinery

Having explained the minimal simulation and the evaluation process, this section describes all the other components of the evolutionary machinery used to produce controllers that could successfully perform the T-maze task. It begins with a description of the type of neural networks that were used, and goes on to explain the encoding scheme, genetic algorithm and genetic operators.

Neural networks

Evolving controllers were recurrent networks of 10 neurons. The number of links to a neuron from other neurons, up to a maximum of 3, were genetically determined (see below). To update the state of the network at each iteration, the input activity A_j of each of the $j = 1\ to\ 10$ neurons in the network was calculated according to the simple weighted sum of equation 2.1

$$A_j = \sum O_i w_{ij} + I_j$$

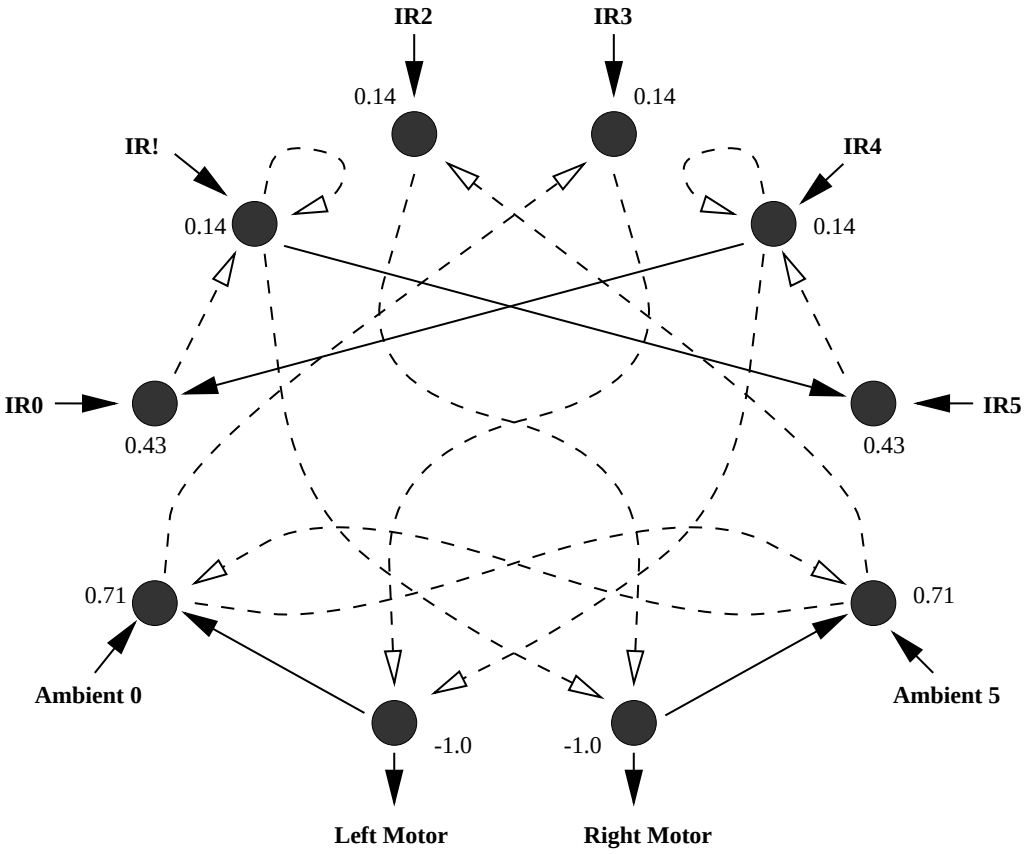

Figure 5.7: *A typical evolved network. The solid arrows are excitatory links and the dashed arrows are inhibitory links; exact weight values are not shown. Threshold values appear next to each neuron.*

where O_i was the output from the i_{th} neuron, w_{ij} was the weight on the connection from the i_{th} neuron to the j_{th} neuron, and I_j was any external input to the j_{th} neuron from outside the network. After the input activity of every neuron in the network had been calculated, the output O_j of each of the $j = 1$ *to* 10 neurons in the network was calculated. If the j_{th} unit was *not* a motor neuron then its output O_j was calculated according to equation 2.3

$$O_j = \begin{cases} 0 & A_j < t_j \\ 1 & A_j \geq t_j \end{cases}$$

and if the j_{th} unit *was* a motor neuron then its output O_j was calculated according to equation 2.4

$$O_j = \begin{cases} -1 & A_j < t_j - 1 \\ A_j - t_j & t_j - 1 \leq A_j \leq t_j + 1 \\ 1 & A_j > t_j + 1 \end{cases}$$

where, in both cases, t_j was a threshold constant associated with the j_{th} neuron. For all neurons, threshold constants were real numbers in the range ± 1.0 and weights on links were real numbers in the range ± 2.0.

Figure 5.7, a diagram of a typical evolved neural network, shows how sensor value inputs were applied to networks, and how motor values were output. All sensor values were normalised in the range 0 to 1 and motor outputs were multiplied by a factor of 10 to give motor signals in the range

$\pm$8cm per second. The network, sensor values and motor outputs (in fact the entire simulation) were updated the equivalent of 10 times a second.

Encoding scheme

A direct encoding scheme was used with a simple one to one mapping between genotype and phenotype. Since the task was bilaterally symmetrical, evolving networks were also forced to be bilaterally symmetrical by encoding the parameters for only half of the network and reflecting it across the midline[1]. Since each network contained ten neurons, therefore, each genotype consisted of only 5 fields of 28 bits, one for each neuron of the left hand side of the network. The neurons on the right hand side of the network were the exact mirror image of those on the left hand side. Each gene was itself divided into fields. The first 4 bits of each gene, a binary number between 0 and 16, defined the threshold of that neuron by normalising between ± 1. The next 3 sets of 8 bits defined the three possible links to that neuron from other neurons in the network: the first 4 ascribing one of 16 possible values for the weight of the link between ± 2 and the next 4 bits defining which of 16 neurons the link was from. Because there were only 10 neurons in total in the network, if a link indexed a non-existent neuron, then it did not connect, thus placing the number of links to a neuron under genetic control.

Genetic algorithm and genetic operators

The genetic algorithm was a steady-state distributed genetic algorithm (Collins and Jefferson 1991) with a population of 100 individuals arranged on a virtual 10 by 10 grid. At each iteration, a random location was chosen on the grid and a breeding pool constructed from the nine individuals of the 3 by 3 square centred on that location. Two probabilistically fit parents were chosen from this breeding pool according to a linear rank-based selection procedure, and an offspring constructed by a process of crossover and mutation. This offspring then replaced a probabilistically unfit member of the same breeding pool according to an inverse linear rank-based selection procedure. Single point crossover was applied with probability 0.7 and the expected number of mutations per genotype, according to a Poisson distribution, was 2. At each offspring event, not only was the offspring's fitness evaluated, but both parents were re-evaluated as well.

5.3 Experimental results

Figure 5.7 shows a typical example of the sort of neural network that consistently evolved within around 1000 generations (where a generation was taken to be 100 offspring events). This is the simulated equivalent of $300 \times 15 \times 10 \times 100 = 45000000$ seconds or over 17 months of continuous real-world evolution, and takes around 4 hours to run as a single user on a SPARC Ultra. The network reliably achieved near-optimal fitness within the simulation. In order to see whether it would successfully transfer across the reality gap, the network was downloaded onto a Khepera robot and tested as to its ability to perform the task in the real world. Sixty different trials were performed one after another, twenty in each of three different widths of corridors, with the light on the left for ten trials and the light on the right for the other ten. The consequent robot behaviours

[1] For a justification of why symmetry was enforced rather than allowed to evolve, see section 2.3.

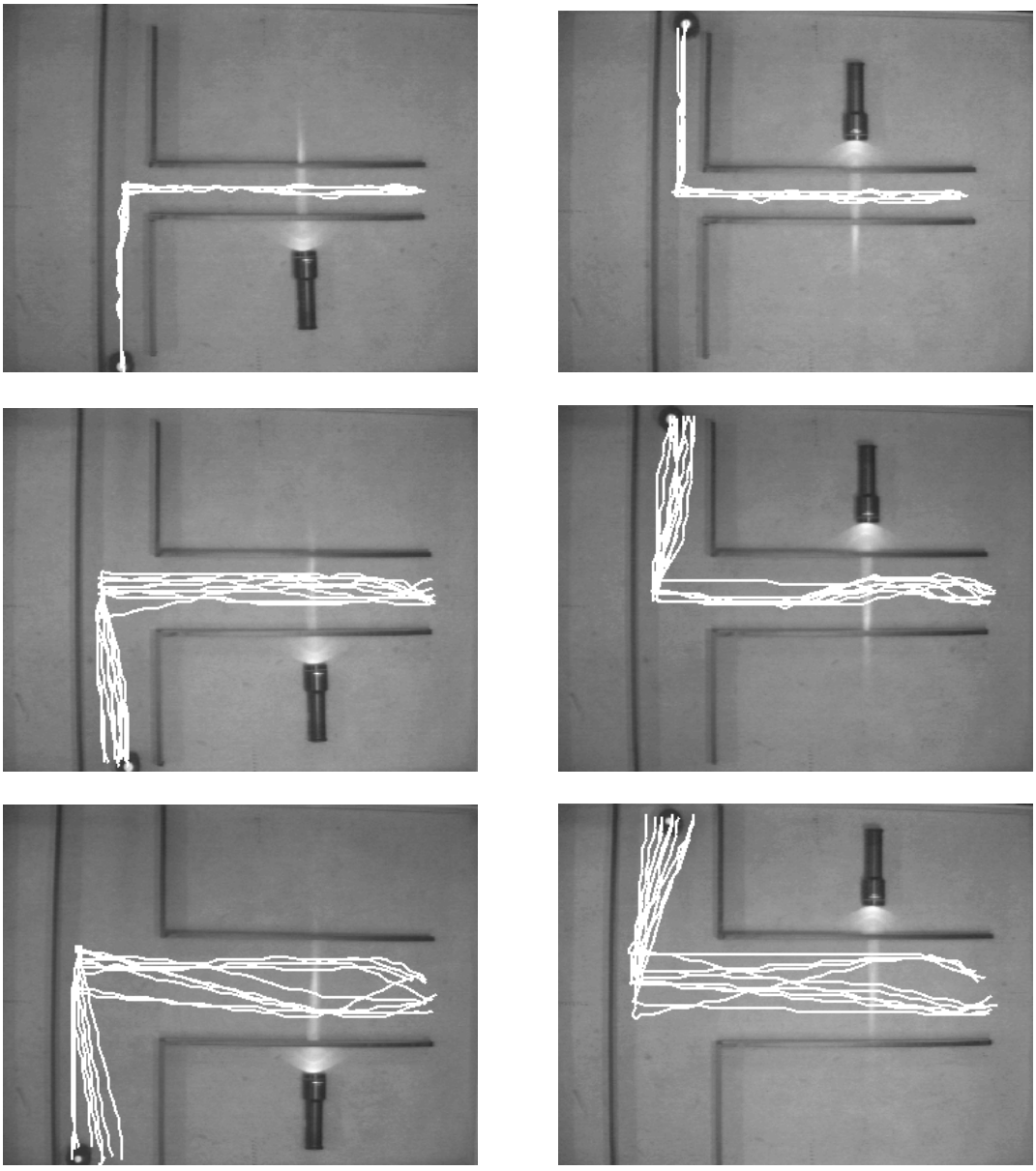

Figure 5.8: *These six pictures together show the paths taken by a Khepera robot in sixty consecutive trials of the control architecture shown in figure 5.7. These sixty trials were performed in consecutive batches of ten, and each picture shows ten trials for a particular corridor width and torch orientation. The pictures were created using an overhead camera, a videodisc, and simple computer vision techniques to find the position of the robot in each frame.*

were filmed from above so that the exact path taken by the Khepera on each trial could be extracted using basic image-processing techniques and overlayed upon aerial views of the set-up. The results of this process are the six images of Figure 5.8.

In the top pair of images, the corridor is only 11cm wide and the paths taken by the Khepera on all twenty occasions are tightly constrained. In the second pair of images, where the corridor is 18cm wide, and especially in the bottom pair of images, where the corridor is 23cm wide, the paths taken by the Khepera are less constrained. The Khepera still turns the correct way at the T-junction, however, even though on several occasions it must turn through greater than ninety degrees in order to do so. Note that the path taken in most cases was near-optimal, and that in every case the task was performed satisfactorily: the criterion put forward in section 3.2 for a control architecture to successfully transfer from simulation to reality.

5.4 Comments

The experiments reported in this chapter represented the first real test for the theory and methodology of minimal simulations laid out in chapter 3.3. As such the minimal simulation could, in retrospect, have been made simpler. In particular, the model of how the infra-red sensors returned values within an infinite corridor could have been made both faster-running and easier to construct by using a few more look-up-tables in place of the somewhat arbitrary mathematical model.

The infra-red sensor model does, however, provide a convincing demonstration of a point first made in section 3.4.1: when building a model of the base set aspects of real-world controller input, it is not necessary to model these aspects for every possible position of the robot within its environment. Put simply, if the way in which robot sensors respond in some areas of the environment is harder to model than in other areas, then we may treat sensor responses as implementation aspects in the harder-to-model areas, and only allow evolving controllers to rely on how they respond in the easy-to-model areas. This makes the job of building a simulation much easier since it can vastly reduce the amount of complex modelling that needs to be done. However, it is important to make sure that the base set aspects of controller input within the simulation are comprehensive enough to allow reliably fit controllers to evolve. If we are not careful we may effectively exclude so many real-world features from the simulation that what we are left with is insufficient for successful behaviour.

Chapter 6

A minimal simulation of the gantry robot

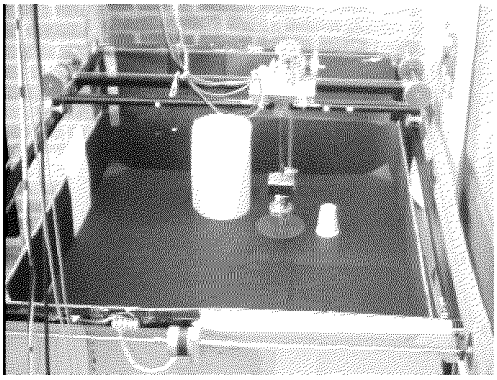

Figure 6.1: *The gantry arena, with obstacles.*

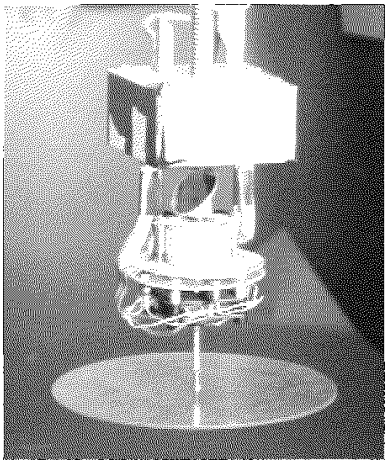

Figure 6.2: *A close up of the gantry robot.*

This section describes experiments in which neural network controllers were evolved for the gantry robot. The gantry, shown in fig 6.1, was developed for research into the evolution of visually guided behaviours, and has been specifically designed so that control architectures can be tested automatically and safely in a highly controlled manner (Husbands, Harvey, Jakobi, Thompson, and Cliff 1997). It is best thought of as a hardware simulation of a small wheeled mobile robot with a camera placed on top.

Figure 6.2 shows a close-up of the robot. A camera points vertically downwards at a 45^o inclined mirror to return a view from the robot looking straight out horizontally at the environment. The mirror is attached to a stepper motor that enables it to rotate around the vertical axis under computer control and a dedicated vision PC then rotates the image array in software so that 'down' in the picture corresponds to 'down' in reality. The image array available for use by evolving control architectures is therefore equivalent to that produced by a camera pointing outwards along the horizontal component of the mirror's orientation. The gantry frame from which the robot is suspended is connected to two further stepper motors that together allow the entire robot assembly to move in any horizontal direction within a rectangular arena (see figure 6.1).

All three stepper motors are controlled by a single board computer (SBC) that is controlled, in turn, by a dedicated brain PC running the control architecture software. The brain PC sends commands to the SBC in the form of left and right wheel speeds, as if the gantry were a wheeled mobile robot. The SBC then calculates and issues stepper motor pulses so that the gantry moves in the appropriate fashion. From the point of view of control architectures running on the brain PC, therefore, the gantry robot behaves exactly as a small wheeled mobile robot, controlled via the SBC, with a camera on top whose image is accessed via the vision PC.

Harvey, Husbands, and Cliff (1994) report experiments in which both neural network control architectures and the visual morphologies of their inputs were evolved side by side to perform a simple shape recognition task: discriminating a triangle from a square and guiding the robot towards it. After several generations, which took approximately 36 hours to perform in the real world, control architectures evolved that were able to perform the task. These controllers were around 80% reliable within certain constrained sets of lighting conditions (Husbands 1997): if the blinds of the laboratory were opened during the day, or if the overhead lighting was not on in the right way, they failed. In order to remedy this sensitivity to differing lighting conditions a set of lamps were strung up above the gantry, each turning on and off at different frequencies, to provide extreme real-world noise for evolving controllers to cope with. The previously fit controllers failed completely when the 'disco lights', as they are known at Sussex, were switched on. As yet, no new controllers have been evolved on the gantry using real-world evolution that are able to cope with the extra uncertainty that these lights provide. Evolving reliably fit control architectures in a minimal simulation, therefore, and seeing whether they were able to perform the task satisfactorily in the real world environment with the 'disco lights' switched on, provides a good test of the theory and methodology put forward in chapter 3.

6.1 The minimal simulation

In the experiments reported in (Harvey, Husbands, and Cliff 1994) both the neural network control architectures and the morphology of their visual inputs were genetically determined. In the simulation experiments reported here, a different type of control architecture was used (see below), although both neural networks and the visual morphology of their inputs were again genetically determined. The main difference between the two, as far as a simulation was concerned, is that in (Harvey, Husbands, and Cliff 1994), each visual input to the neural network consisted of the average grey-level value of a genetically specified circular sub-region of the camera image, whereas in the experiments reported here, each visual input consisted of the grey-level value of exactly one genetically specified pixel of the camera image (Figure 6.4). In fact, these are not so different with respect to a simulation, since the average value of each circular visual field in (Harvey, Husbands, and Cliff 1994) was just the average value of 25 randomly sampled pixels from within the field. A simulation of either, therefore, must contain a model of how specific pixels of the camera image acquired values in response to the orientation and position of the robot within its environment. Below, the step-by-step framework of section 3.5 is used to explain how the minimal simulation was constructed.

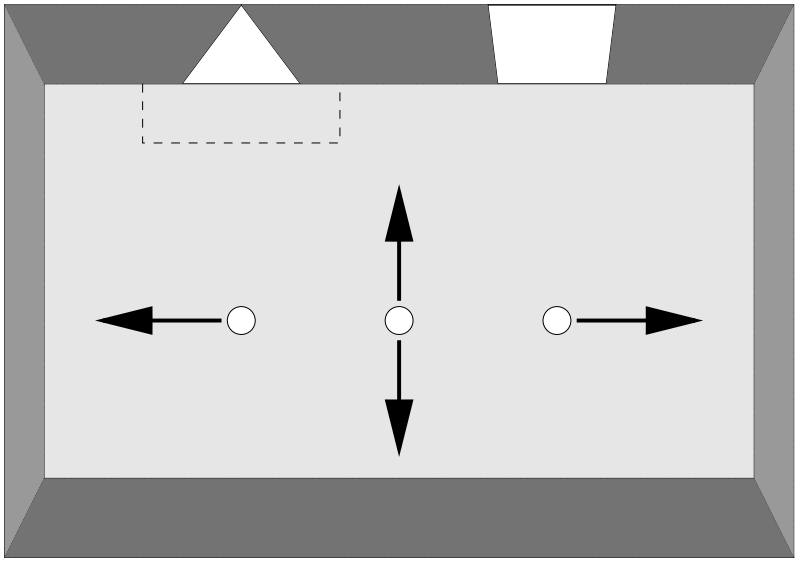

Figure 6.3: *A diagrammatic view of the gantry arena from above showing the four possible starting positions of the gantry robot. The dashed line in front of the triangle marks the area that the gantry must reach in order for a trial to count as a success for testing purposes.*

Precisely define the behaviour

The environment consisted of a rectangular arena, 150cm by 100cm, with 22.5cm high walls painted black. Stuck onto one of the long walls were a near-square (20cm wide by 22.5cm high) and an equilateral triangle (20cm wide by 22.5cm high), both of which were cut from white paper. Starting from each of four different positions and orientations (see figure 6.3), evolving individuals had to steer the gantry robot towards the triangle while ignoring the square.

Identify the real-world base set

Whether or not the controller performs the behaviour is a function of the robots path within the arena and with respect to the triangle. The features of the world that can affect this path are those that make up the causal pathway from controller output to movement of the robot within its environment. These include the way in which controller output affects the movement of the wheels, and the way in which wheel motion affects the position of the robot within the rectangular arena and with respect to the triangle.

Build a model of the way in which the members of the base set interact with each other and react to controller output (when the robot is performing the behaviour).

The model of the way in which the gantry robot moves in response to motor signals was adapted from the movement model for the Khepera robot explained in Chapter 5. The simulation was again updated at a rate equivalent to ten times a second and the same look-up-table was used but with different constants to update speed, orientation and position variables at each iteration of the simulation. The radius of the virtual robot (that the gantry robot is a hardware simulation of) is 15cm and the constant multiplied by the motor signals to give the current speed of the robot is 4.17 cm per motor unit per second. In addition there was also a momentum term, m, such that at

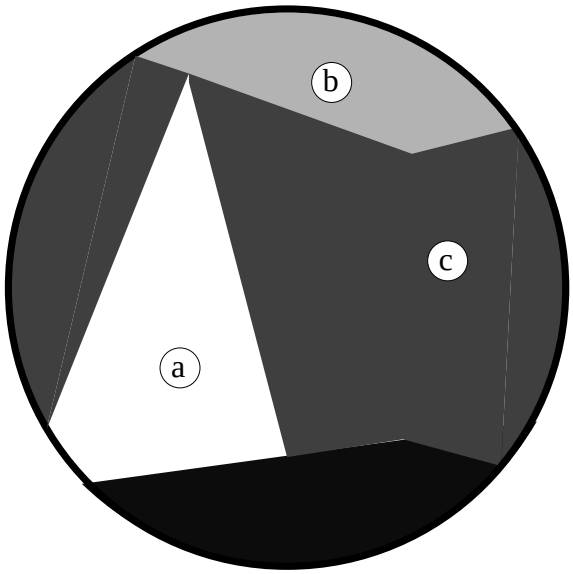

Figure 6.4: *A typical image returned by the camera of the gantry robot. The robot is facing the corner of the arena and the triangle can be seen on the left. The white circles labelled a, b and c are examples of pixels that project onto the triangle, ceiling and wall respectively. Pixel a will return a value between 14 and 15, pixel b will return a value between 0 and 15 and pixel c will return a value between 0 and 13. In the experiments reported below, each visual input was made up of exactly one pixel whose coordinates within the camera image was genetically determined.*

each iteration, the increment δv to each wheel speed v in terms of the required wheel speed u was:

$$\delta v = \frac{u - v}{m} \tag{6.1}$$

This momentum term was added for the simple reason that in the case of the gantry, momentum plays a significant role since it is a heavy robot which takes time to slow down and speed up. In the case of the Khepera, the robot is small enough and light enough that momentum effects can be regarded as modelling inaccuracies, and can be coped with by reliably fit control architectures that are base set robust (see 3.4.2). At every iteration, a random deviate in the range ± 0.2 cm per second was added to each wheel speed to approximate the noise inherent in the way the gantry robot moves.

Build a model of (enough of) the way in which the members of the base set affect controller input (when the robot is performing the behaviour).

Under the 'disco lights' suspended above the gantry, the values returned by pixels of the camera-image vary widely both with respect to time, and with respect to the direction of the camera. Even if we know the exact location within the arena that a particular pixel projects onto, there is not that much that can be said about exactly what the value of that pixel will be. However, there are a few general things that hold true except in exceptional circumstances: if a pixel projects onto a wall but not onto a shape then it returns a value within the range 0 to 13, if a pixel projects onto either the triangle or the square then it returns a value between 14 and 15, and if a pixel projects onto either the floor or the ceiling of the arena it returns a value between 0 and 15. Since these facts

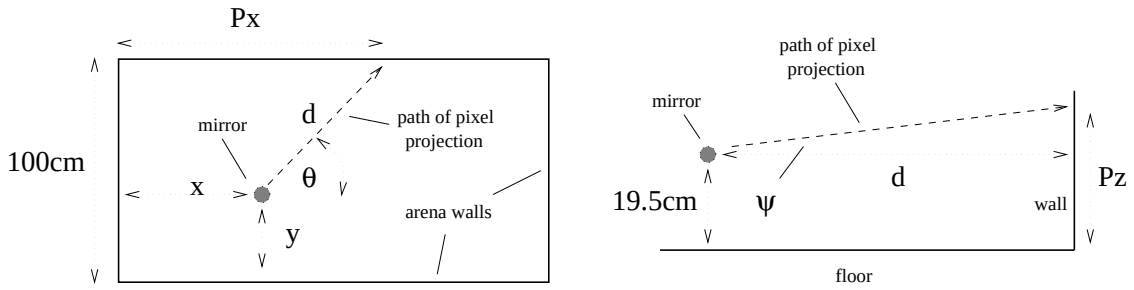

Figure 6.5: *The left-hand picture shows the gantry arena as seen from above; if the horizontal angle at which a pixel projects from the mirror onto the back wall is* θ*, then* $Px = x + \frac{100-y}{tan\theta}$ *and* $d = \frac{100-y}{sin\theta}$*. The right-hand picture shows a cross-section of the gantry arena; if the vertical angle at which a pixel projects from the mirror onto a wall is* ψ*, then* $Py = d \times tan\psi + 19.5$*.*

about pixel values within the 'disco light' environment are almost always the case, and since they are enough to distinguish the white triangle and square from the black walls of the arena (for those pixels that project onto a wall of the arena), they were all we needed to model.

To work out the location in the arena that a particular genetically specified pixel projected onto was done using simple trigonometry. Look-up-tables were employed in place of the computationally expensive standard C library functions of *cos*, *sin* and *tan*. Each table contained 360 values covering 360^o. In addition there was a finer-grained look-up-table for *tan* containing 200 values, one for every 0.1^o between 0^o and 20^o.

After rotation by the vision PC, the image available to evolving control architectures on the gantry robot is a circular portion of a two dimensional pixel array, 40 pixels in diameter and with an angle of acceptance of around 50^o (see Figure 6.4). The horizontal and vertical angular offsets of any particular pixel from the orientation of the robot were calculated from its x and y coordinates within the image, and were then used to work out the horizontal and vertical angles at which the pixel projected out from the gantry robot's mirror relative to the fixed arena environment. Since the coordinates of the robot's position within the arena were always known, and the height of the mirror above the floor of the arena was fixed (around 19.5cm), the exact horizontal and vertical coordinates of the spot that any particular pixel projected onto could be easily worked out. Firstly, the simulation established which of the four walls of the arena a particular pixel would project onto if the vertical angle was in the correct range, and calculated the horizontal coordinate of the pixel projection onto that wall. Secondly, the vertical coordinate of the pixel projection onto the wall was calculated. The way this was achieved is demonstrated in figure 6.5. For calculations of *Px* (the horizontal coordinate of the point a pixel projects onto), the course-grained *tan* look-up-table was used, and for calculations of *Pz* (the vertical coordinate of the point a pixel projects onto), the fine-grained *tan* look-up-table was used. This is because ψ will always be a small angle somewhere between 0^o and 25^o whereas θ can be anything between 0^o and 360^o.

Having worked out *Px*, *Pz*, and the relevant arena wall, the actual value attributed to a particular pixel depended on one of three possible scenarios. Either the pixel projected onto the floor or ceiling, in which case it returned a value between 0 and 15, or it projected onto a wall but not onto the triangle or square, in which case it returned a value between 0 and 13, or it projected onto the triangle or square, in which case it returned a value between 14 and 15. The *ways in which*

values were returned from within these intervals are described below (in the section on ensuring that controllers are bas set robust). If *Pz* was less than 0cm or *Pz* was greater than 22.5cm then the pixel was judged to have projected onto either the ceiling or the floor. If *Pz* was between 0cm and 22.5cm it was judged to have projected onto a wall. If the wall in question was the one with the triangle and the square on it, then simple geometric relationships between the coordinates of the pixel projection point and the vertices of the two shapes were used to find if the pixel projection point lay inside either of the shapes. At every iteration, random deviates (generated by a simple multiplicative congruential random number generator (Press, Vetterling, Teukolsky, and Flannery 1992)) in the range ± 1.2 grey-scale units were added to each pixel value. This corresponds roughly to the noise present in the real world over and above that produced by the disco lights.

Design a suitable fitness Test

The fitness function returned the average value scored by an individual in a total of eight fitness trials, each trial lasting a maximum of twenty simulated seconds. For the first set of four trials, the triangle was on the left and the square was on the right, and for the second set of four trials, the triangle was on the right and the square was on the left. For both sets, the robot was started at each one of the four starting positions shown in Figure 6.3 in turn. At the end of each trial, when either the time had run out or the robot had hit a wall, the fitness function returned $100 - d$ as the fitness score, where d was the distance from the centre of the robot to the centre of the triangle.

Ensure that evolving controllers are base set exclusive

As reported above, the way in which pixel values were returned within the relevant intervals was treated as an implementation aspect of the simulation and varied from trial to trial according to the methodology outlined in Chapter 3.3. This ensured that control architectures that had evolved to be reliably fit within the simulation worked independently of the way in which actual pixel values arose - as long as they arose within the specified intervals - and therefore that they were robust to the ‘disco lights’.

At the beginning of each trial, one of three ways of generating pixel values within the appropriate intervals was chosen:

1. Each pixel returned a different random value within the appropriate interval and values varied randomly over time. This meant that whatever the behaviour of the robot, values could change. The average time interval between changes in value for any particular pixel was taken from a Poisson distribution with an average length of 2 simulated seconds.

2. Each pixel returned a different random value within the appropriate interval and values only varied in response to changes in robot-orientation. This meant that if the robot proceeded in a straight line, or remained still, then pixel values remained steady. If the robot turned, then pixel values could change to new random values: angular distances between changes in value for any particular pixel averaged 25^o and were uniformly distributed between 0^o and 50^o.

3. Each pixel returned the same random value within the appropriate interval. Values for each interval were kept constant throughout the trial.

In this way, reliably fit controllers were forced to employ strategies that depended solely on the intervals that pixel values fell into and not on the specific values themselves.

Ensure that evolving controllers are base set robust

Using a knowledge of the inaccuracy of the model, various aspects were varied from trial to trial in order to ensure that reliably fit control architectures were base set robust (see Section 3.4.2). This was especially important with a robot such as the gantry which is extremely noisy and imprecise in its operation. In particular, the mirror that reflects the horizontal image up into the camera was not set at exactly 45^o and was slightly warped. This meant that objects appeared differently depending on where they were in the camera image, and that as the robot approached an object its image would deform and distort, appearing to move upwards. Because of this:

- A vertical angular offset of between -1^o and -8^o was produced at the beginning of each trial. This was then added to the vertical angle of projection of every pixel throughout the trial.
- A horizontal angular offset of between $\pm 3^o$ was produced at the beginning of each trial. This was then added to the horizontal angle of projection of every pixel throughout the trial.
- The horizontal coordinates (with respect to the wall) of the four corners of the square and the three corners of the triangle were offset by a random amount within the range $\pm$5cm throughout each trial.

The stepper motors moved the gantry robot along rollers using drive-chains. These rollers slid rather than rolled along their rails (due to a design fault), with more friction in some places than others, and the drive belts were loose so that rapid sequences of motor commands could get lost in the extra 'slop'. Because of this, the robot could only approximate travelling at a constant speed, and neither accelerated nor braked evenly in response to motor commands. It would often seize completely half way through a run. In order that reliably fit individuals evolved to cope with these problems:

- The momentum term, *m*, of equation 6.1 was randomly set at the beginning of each trial to a value between 1 and 4.
- Random offsets of between ± 0.5cm per second were generated at the beginning of each trial, and added to required wheel-speeds during position update calculations.

Together these random variations ensured that reliably fit control architectures were able to cope with a wide variety of slightly different robot-environment interaction models. Included in this range were models that involved misshapen and mal-aligned mirrors as well as noisy and unpredictable motors - such as the model instantiated by the real gantry robot.

6.2 The evolutionary machinery

Although the evolutionary machinery (controllers, encoding scheme, genetic algorithm and genetic operators) used in (Harvey, Husbands, and Cliff 1994) was initially reimplemented for the experiments described here in order to provide a direct comparison, it was later abandoned; reliably fit individuals failed to evolve run after run. In the simulation, evolving controllers had to cope with a whole variety of slightly different base set aspects, rather than just the one base set present in the real-world situation. The implication was that the evolutionary machinery used in the original experiments was just not capable of producing the level of robustness necessary to

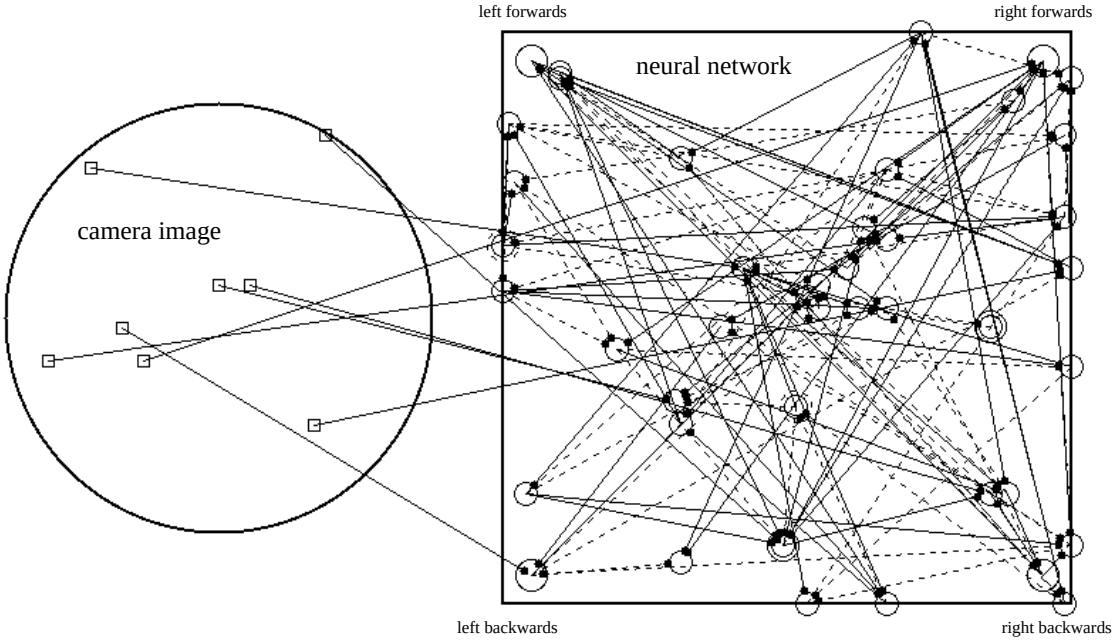

Figure 6.6: *An example of a typical neural network evolved for the triangle/square discrimination task. On the left, the circular camera image, eight pixels of which have been genetically specified as inputs to the neural network. On the right, the square box contains the neural network itself. Solid lines denote excitatory connections and dashed lines denote inhibitory connections. The slightly larger units in each of the four corners are motor neurons.*

cope with this extra uncertainty. The neural networks, encoding scheme, genetic algorithm and genetic operators that were used instead are described below.

Neural networks

Functionally, the neural networks used in the experiments reported here were very similar to those used in the T-maze experiments of chapter 5. To update the state of the network at each iteration, the input activity A_j of each of the $j = 1\ to\ N$ neurons in the network was again calculated according to the simple weighted sum of equation 2.1

$$A_j = \sum O_i w_{ij} + I_j$$

where O_i was the output from the i_{th} neuron, w_{ij} was the weight on the connection from the i_{th} neuron to the j_{th} neuron, and I_j was any external input to the j_{th} neuron from outside the network. After the input activity of every neuron in the network had been calculated, the output O_j of *all* (including the motor neurons) of the $j = 1\ to\ N$ neurons in the network was calculated according to a version of equation 2.3

$$O_j = \begin{cases} 0.05 & A_j < t_j \\ 1 & A_j \geq t_j \end{cases}$$

where t_j was a threshold constant associated with the j_{th} neuron. The value of 0.05 replaced 0 as the 'off' state of each neuron so that self-excitatory feedback loops would be forced to saturate. Weights on links were in the range ± 2 and thresholds were in the range 0 to 1.

All sensor input values were normalised in the range 0 to 1. Motor signals were calculated from the output values of the four larger corner neurons (see figure 6.6) according to the relation

$signal = 2 \times (O_1 - O_2)$, where O_1 and O_2 were the output values of the appropriate forwards and backwards neurons. The whole network, including inputs and outputs, was updated at a speed of 10 times per second (or the simulated equivalent of a second).

Encoding scheme

The encoding scheme was a version of the spatially determined encoding scheme described in section 2.3 which allows genotypes to grow under genetic control with a minimum amount of phenotypic disruption. Figure 6.6 shows the two dimensional space in which development took place. Apart from the position of the four motor neurons which were fixed (see figure), the position of each neuron within the space was genetically determined. The links *to* each neuron in the network were genetically specified by way of target positions that the links ideally originated *from*. The nearest neuron to each link's target position, within a radius of 1/10th of the width of the space, was alloted as the originator of the link. If no neurons lay within this radius then the link failed to connect. In addition, every neuron in the network could potentially receive input from a genetically specified pixel of the camera image.

Each gene was 15 integers long, each integer lying between 0 and 99 and specifying its corresponding parameter through a simple linear mapping. Apart from the first four genes, which specified the characteristics of the four positionally fixed motor neurons, the first two numbers of each gene specified the x and y coordinates of the corresponding neuron's position within the developmental space. The next number specified whether a neuron received input from a pixel of the camera image or not, with a probability of 1 in 4, and the next two numbers specified the x and y coordinates within the camera image of any pixel input. The sixth number of each gene specified the threshold, between 0 and 1, of the corresponding neuron. The last nine numbers specified the characteristics of up to three possible links *to* the relevant neuron *from* other neurons in the network: three number per link. The first two of these three numbers encoded the link's target position within the developmental space, and the third number specified the weight.

Genetic algorithm and genetic operators

The genetic algorithm used in the experiments was extremely simple. After testing every member of a population of 100 individuals, the fittest 25 were used to produce the next generation by randomly picking parents from within this 25 and producing offspring until the new population was full. Single-point crossover was applied with a frequency of 0.7 and the expected number of mutations per genotype, according to a Poisson distribution, was 1. There was a probability of 0.02 at each offspring event that a random gene would be introduced into the offspring genotype, as well as a probability of 0.02 that an already existing gene would be deleted.

6.3 Experimental results

Figure 6.6 shows a typical example of the sort of network that evolves to be reliably fit within the simulation. This particular network is the result of around 6000 generations of the genetic algorithm (around 12 hours as a single user on a SPARC Ultra), which is the simulated equivalent of $6000 \times 100 \times 8 \times 20 = 96000000$ seconds, or over 3 years worth of real-world evolution. When placed in one of the four starting positions in the arena, the network initially causes the robot to

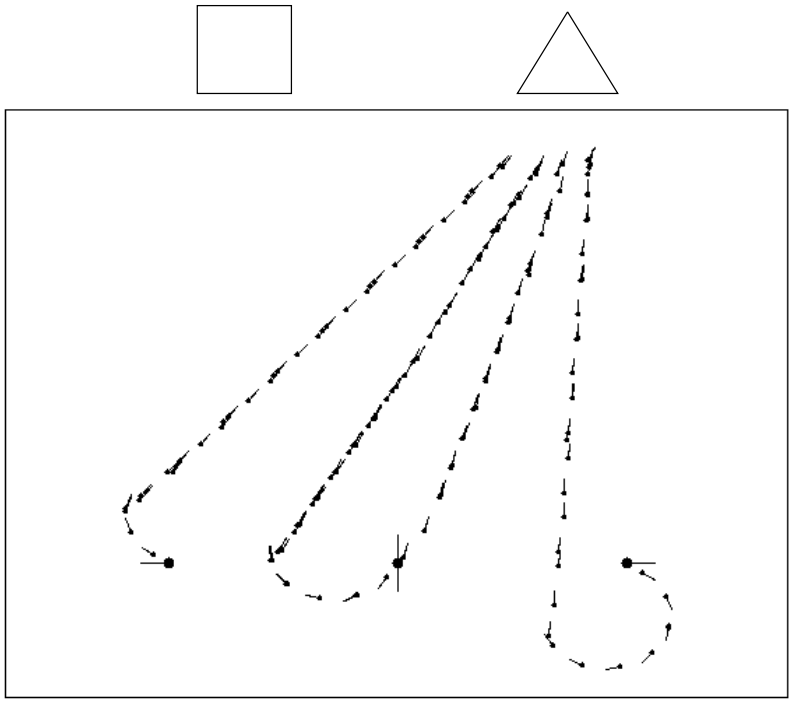

Figure 6.7: *Position data taken twice a second from the gantry robot during sixteen consecutive trials with the triangle on the right and the square on the left. Each set of position coordinates is displayed as a black dot with a small black line projecting along the forward orientation of the robot.*

turn in a tight circle clockwise. If the square comes into the view of the camera, the rotational speed of the robot actually increases until the square is out of view. When the triangle hoves into view, the robot 'locks on' and precedes directly towards it, adjusting its course as it goes.

In order to see whether it would cross the reality gap, the network was downloaded onto the gantry, and tested continuously[1] and automatically on the triangle/square task in the real world under full disco lighting. In total, 200 trials were performed: 100 for the triangle on the left and the square on the right, and 100 for the triangle on the right and the square on the left. At the beginning of each trial the robot was started in one of four different starting positions, corresponding to those of the simulation, and these were run through in cycle from trial to trial. On each trial, the robot was automatically judged to have successfully achieved the task if, by the end of the trial, it was stationed within a rectangular area extending about 10cm either side of the triangle and 15cm out into the arena (see Figure 6.3). Inspection revealed that this automatable criterion corresponded well with more subjective notions of success and failure on the task.

In total, the robot successfully navigated its way towards the triangle while avoiding the square 195 times out of 200. With the triangle on the right and the square on the left the robot performed the task successfully 98 times out of 100. With the triangle on the left and the square on the right, the robot performed the task successfully 97 times out of 100. Figure 6.7 plots position data taken twice a second from the gantry robot during sixteen consecutive trials, four from each of the four starting positions.

[1] In practice, because of the propensity of the mechanics of the gantry robot to cease and the software controlling it to crash, the testing procedure had to be watched continuously, and restarted (from where it had crashed) on a number of occasions.

Of the two failures with the triangle on the right, one occurred when the gantry rails were being polished (to try and prevent the motors from jamming) and the lights were temporarily obscured by the author's body. The other failure is harder to account for, since the gantry robot just headed off into a wall under otherwise unremarkable circumstances. This may have been due to freak noise, but may also have been due to a mechanical or software error. All three failures with the triangle on the left occurred from the same starting position furthest from the triangle and in each case the circumstances were similar. Having turned away from the wall, the robot failed to lock on to the triangle but continued spinning on the spot. It would spin past the square, past its original starting orientation, and back round to face the triangle. In two out of three of the cases it then locked onto the triangle, and started to move directly towards it, running out of time before it reached the success zone. In the third case it failed to lock on again, and ran out of time before it could spin right round to face the triangle for a third attempt. In all three cases, if more time had been allowed, the robot would almost certainly have reached the target.

6.4 Comments

The minimal simulation of chapter 5 demonstrated one of the two main points of section 3.4.1: when building a model of the base set aspects of real-world controller input, it is not necessary to model these aspects for every possible position of the robot within its environment. The minimal simulation used in this chapter demonstrates the other main point: it is not actually necessary to model *all* of the base set aspects of real-world controller input for *any* position of the robot within its environment. Provided sufficient base set aspects are modelled, the rest can be treated as implementation aspects and varied from trial to trial. Thus the minimal simulation described above only modelled the intervals that real-world pixel values could fall within. The *ways in which* values fell within these intervals in the real world were treated as implementation aspects and varied from trial to trial. This vastly reduced the amount of modelling necessary to create a simulation capable of evolving controllers that could cross the reality gap.

Chapter 7

A minimal simulation for a complex motor behaviour

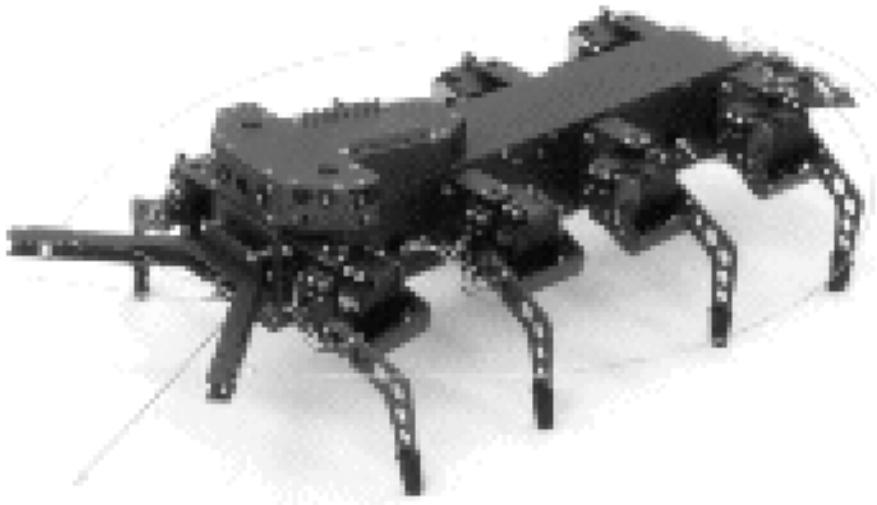

Figure 7.1: *The Octopod robot.*

This chapter describes experiments in which neural network control architectures were evolved for an octopod robot. The robot, shown in figure 7.1 is around 30cm long and has 4 infra red sensors that point ahead and to the side, various bumpers and whiskers, and ten ambient light sensors positioned strategically around the body. Each of the robot's eight legs is controlled by two servo motors, one for movement in the horizontal plane, and one for movement in the vertical plane, which means that the robots motors have a total of sixteen degrees of freedom.

The aim of the experiments was to evolve neural network control architectures that would allow the robot to wander around its environment avoiding objects using its infra-red sensors and backing away from objects that it hits with its bumpers. This is a hard behaviour to evolve when one considers that in order to achieve any sort of coherent movement the controller has to control not just one or two motors in a coordinated fashion but sixteen. Moreover it is an extremely difficult set-up to simulate using traditional techniques since the physical outcome of sixteen motor

movements is rarely predictable in all but the simplest cases. The evolution of this behaviour in a minimal simulation, therefore, provides essential evidence that complex motor behaviours can be evolved in simulations built according to the theory and methodology put forward in Chapter 3.3.

The minimal simulation used to evolve controllers for the octopod is described in Section 7.1. The rest of the evolutionary machinery, including the neural networks, the encoding scheme, the genetic algorithm and genetic operators is described in section 7.2. Experimental results are put forward in section 7.3 and finally, in section 7.4, some comments are offered on the chapter as a whole.

7.1 The minimal simulation

According to received wisdom, simulating something as complex from an actuator point of view as an eight-legged robot is hard. The problems arise from the fact that sixteen motors all moving at the same time and interacting with each other in the real world rarely induce movement in the robot that is easy to model and often produce completely unpredictable movement that is best looked at as stochastic. What happens when two legs clash, for instance? Or when the belly of the robot is on the ground but the legs attempt to push the robot forwards? Or when 4 of the legs attempt to push the robot forwards and 4 of the legs attempt to push the robot backwards? Clearly any simulation that sets out to model *all* of the dynamics of the system will involve vast quantities of pain-staking empirical measurement and research into friction-coefficients, the power of each motor, the range of possible movement of the robot and so on. If the only simulation in which we could evolve autonomous walking behaviour for the real robot was of this type then the simulation would be so complicated that it might indeed be simpler to evolve controllers on the real thing.

Happily we do not need to come close to modelling all of the possible dynamics of the robot in order to build a satisfactory minimal simulation. The key is to realise that those portions of the possible dynamics of an octopod robot which are difficult and complicated to model (the vast majority) are precisely those that are *not* involved in successful walking behaviour. When the octopod robot walks around its environment in an acceptable manner, its legs do *not* clash and its belly does *not* drag along the ground and its legs do *not* pull in different directions. The minimal simulation described below takes full advantage of this fact. The dynamics of the simulated robot match the dynamics of the real robot only when the controller is inducing acceptable, successful walking and obstacle-avoiding behaviour. If a controller does anything else *but* acceptable, successful walking and obstacle-avoiding behaviour then the simulation falls woefully short of modelling what would actually happen in the real world. Since a controller that performs the behaviour will never take the robot into this region of the dynamics, we do not need to model it.

Precisely define the behaviour.

The aim of the experiments was to evolve octopod-controllers that could walk around the environment, turning away from objects that fell within range of the IR sensors and backing away from objects that touched the front bumpers and whiskers. At the very least, this requires that controllers are able to perform 4 sub-behaviours, each relevant to a particular sensory scenario:

- If an object falls within range of the left-hand IR sensors then the robot must turn on the spot to the right.

- If an object falls within range of the right-hand IR sensors then the robot must turn on the spot to the left.
- If an object hits the front bumpers or whiskers then the robot must walk backwards as fast as possible.
- In the absence of any objects falling within infra-red range or touching the robot's front bumpers or whiskers, the robot must walk forwards in a straight line as fast as possible.

In a cluttered environment there are occasions in which other sensory combinations may occur e.g. objects may fall within range of the left and right IR sensors at the same time. However, these occasions are rare enough in simple environments to grant that controllers which are able to perform each of these 4 simple sub-behaviours are also capable of wandering around their environment satisfactorily without bumping into anything or becoming stuck.

One reason for making this behavioural reduction is that constructing a fitness test that specifically checks for each of the 4 sub-behaviours, one after the other, is actually much easier than constructing a fitness test that checks directly for the more complex global behaviour. We do not need to simulate, for example, the way in which the robot's position within a complex environment gives rise to sensor values. Instead we may test directly for each of the 4 sub-behaviours in turn by clamping the sensor values to fit each of the 4 sensory scenarios and observing the movement of the robot in response. The fitness function was therefore divided into 4 phases: each testing for one of the 4 behaviours outlined above. The order in which each of the 4 phases occurred was random and evolving neural network controllers were not reset in between. This ensured that reliably fit controllers would be able to perform each of the 4 sub-behaviours independently.

Identify the real-world base set

Whether or not the robot satisfactorily performs each of the 4 sub-behaviours is a function of the movement of the robot body in each of the 4 different sensory scenarios. The members of the base set, therefore, are those features of the world that make up the causal pathway from controller output to how the body as a whole moves in response. These include the way in which controller output affects how the legs move, and the way in which the movement of the legs affects the movement of the body as a whole.

Build a model of the way in which the members of the base set interact with each other and react to controller output (when the robot is performing the behaviour).

The overall movement of the robot was described by two variables: one for the speed of the left-hand side of the robot and one for the speed of the right-hand side of the robot. Thus if both sides of the robot moved straight ahead at the same speed then the overall movement of the robot was deemed to be straight ahead, if they moved in different directions but with equal velocity then the robot was deemed to be turning on the spot, and if both sides moved backwards at the same speed then the overall movement of the robot was deemed to be straight backwards.

To model the way in which the robot as a whole moved in response to controller output, therefore, necessitated a model of the way in which each leg responded to controller output, and the way in which the movement of each leg contributed to the overall movement of each *side* of

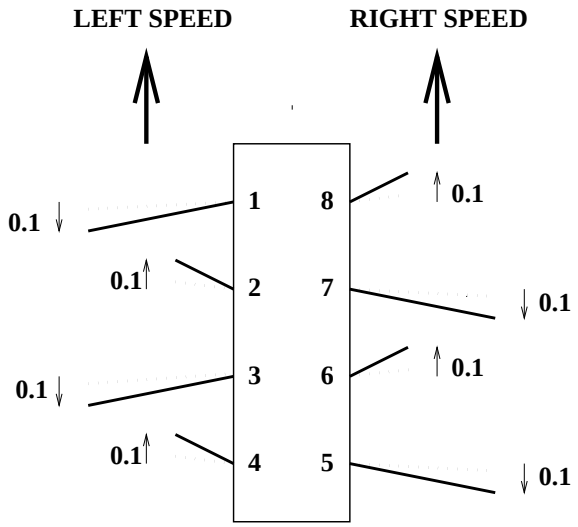

Figure 7.2: *This figure shows diagrammatically how the speeds of the left-hand and right-hand sides of the robot were calculated from the vertical and horizontal positions of the eight legs. For explanatory purposes the length of each leg in the diagram is inversely proportional to its height above the ground so that the long legs are 0.8 as low as they can go and the short legs are 0.2 as low as they can go. Adding up the contributions that each leg makes to the speed of its side we see that the speeds of both the left and the right hand side of the robot work out at* $0.1 \times 0.8 - 0.1 \times 0.2 + 0.1 \times 0.8 - 0.1 \times 0.2 = 0.12$ *forwards*

the robot. However, because of the arguments put forwards in section 3.3.2, it was not necessary to accurately model the way in which *every* motor signal could affect the movement of the robot as a whole, but only those motor signals involved in satisfactory walking forwards, backwards and turning on the spot. The dynamics of the model, therefore, matched those of reality only for those controllers that prevented the body from touching the ground, moved all the legs supporting the robot on each side in the same direction (either all forwards or all backwards depending on whether the robot was supposed to be walking forwards, backwards or turning on the spot), and kept those legs that were not touching the ground as high in the air as possible.

The motor signals to the servo-motors controlling the legs of the octopod robot specify absolute angular positions (relative to the body) that the servo-motors are required to move the legs to. Thus when a new signal is sent to the servo-motor controlling the horizontal or vertical angle of a particular leg, it will move as fast as possible to the new location. In the absence of any new signal, the leg will remain rigid. This process was modelled in the simulation by calculating, on every iteration, horizontal and vertical angular displacements for each leg based on the differences between the angular positions specified by the motor signals and the actual angular positions of the simulated legs. The maximum possible angular speed of each leg was measured very roughly and set in the simulation to be 2π radians per second. Using the horizontal and vertical angles of each leg, a simple look up table provided the approximate position, relative to the robot, that each leg projected onto the ground, and the 4 legs in the lowest positions were assigned as the supporting legs. A simple calculation was then made to see whether the robot's centre of gravity was contained within the polygon subtended by the floor-contact positions of these 4 legs, in which case the robot was deemed to be stable. If it was not, then the robot was deemed to be unstable. Also the average height of these 4 legs relative to the robot body was calculated. If they were low

enough then the robot was deemed to be standing, otherwise it was deemed to be dragging its belly on the ground.

Figure 7.2 shows diagrammatically how the speeds of the left and right-hand sides of the robot, and thus the overall movement of the robot, were calculated from the controller's motor signals. On each iteration, the contribution each leg made to the forwards or backwards movement of its side of the robot was worked out according to a simple calculation. The distance moved by the leg (either forwards or backwards) was multiplied by a figure between 0 and 1 that was inversely proportional to how high in the air the leg was. Thus the higher in the air a leg was, the smaller the contribution its horizontal movement made to the total movement of its side of the robot. The nearer to the ground the leg was, the larger the contribution its horizontal movement made to the total movement of its side of the robot. The contributions that each leg makes were then added up to arrive at a figure for the total movement (either forwards or backwards) of that side of the robot. If both the left and the right side of the robot moved forwards then the robot was deemed to have moved forwards, if both sides moved backwards then the robot was deemed to have moved backwards, if each side moved in different directions then the robot was deemed to be turning on the spot.

Now although this simple model seems to bear no relationship to reality (how can a leg that is in the air contribute to the speed of its side of the robot?), a controller that made maximum use of the dynamics of the model to move the robot around as fast as possible would keep all of the legs that were moving in the wrong direction at any one time as high in the air as possible and all the legs that were moving in the appropriate direction as firmly on the ground as possible. Since penalty terms for both instability and belly-dragging were included in the fitness function (see below), maximally fit controllers remained stable and stood upright at all times, moving all the legs that were supporting the robot on each side in the same direction (either all forwards or all backwards depending on whether the robot was walking forwards, backwards or turning on the spot) and keeping those legs that were not supporting the robot as high in the air as possible.

Build a model of (enough of) the way in which the members of the base set affect controller input (when the robot is performing the behaviour).

The sensor model employed was so simple as to be almost non-existent. The sensors were divided into three groups: the front left and back left IR sensors forming one group, the front right and back right IR sensors forming another group, and the front whiskers and bumpers forming another. In the phase of each fitness test in which there were no objects within sensor range, all sensors were set to background levels for the duration of the phase: 0 for the bumpers and whiskers and 255 for the IR sensors. In the phases during which an object fell within IR range on either the left or right-hand side of the robot, the IR sensor on the appropriate side was set to high (200) for the duration of the phase. In the phase during which an object hit the touch sensors, the front whiskers and bumper were set to high (1), *but only for the first second of the phase*. This simple sensor model provided evolving controllers with enough information about the world to perform the behaviour satisfactorily.

Design a suitable fitness Test

As explained above, each fitness evaluation was divided into 4 phases: one for each of the 4 sensory scenarios. Each of these phases lasted five simulated seconds. At the end of every iteration of the simulation, the fitness of the controller being tested was incremented by a value δ derived from the overall movement of the robot. How this value was calculated depended on the sensory scenario the robot was in at the time:

- If there were no objects within sensor range then δ was the speed of the left-hand side of the robot plus the speed of the right-hand side of the robot.
- If there was an object within infra-red sensor range on the right-hand side of the robot then δ was the speed of the right-hand side of the robot *minus* the speed of the left-hand side of the robot.
- If there was an object within infra-red sensor range on the left-hand side of the robot then δ was the speed of the left-hand side of the robot *minus* the speed of the right-hand side of the robot.
- If an object hit the bumpers then δ (for the duration of this phase of the fitness evaluation) was *minus* the speed of the left-hand side of the robot *minus* the speed of the right-hand side of the robot.

Also on each iteration, if the robot was deemed to be unstable then a small penalty was subtracted from the fitness, and if the robot was deemed to be touching the ground with its belly then a small penalty was subtracted from the fitness.

Ensure that evolving controllers are base set robust and base set exclusive

The fitness test described above was carefully designed so that controllers that evolved to be reliably fit would use only those portions of the simulation dynamics that corresponded closely to the dynamics of the real robot. In fact, these dynamics turned out to be close enough that there was no need to vary the simulation at all in order to ensure that evolved controllers were base set robust. Any differences between simulation and reality were easily accommodated by slop in the definition of satisfactory walking and obstacle-avoiding behaviour. Thus walking behaviour on the real robot might be a little jerkier or quicker than in the simulation, but it was still perfectly adequate walking behaviour.

Likewise, nothing extra was added to the simulation in order to ensure that evolving controllers were base set exclusive. This was for the simple reason that the model of the way in which sensor values arose from the base set was so simple that there was nothing else in the simulation that evolving controllers could come to rely upon.

7.2 The evolutionary machinery

In this section we describe the evolutionary machinery that, together with the minimal simulation described above, was responsible for evolving neural network control architectures that could perform the behaviour satisfactorily in reality.

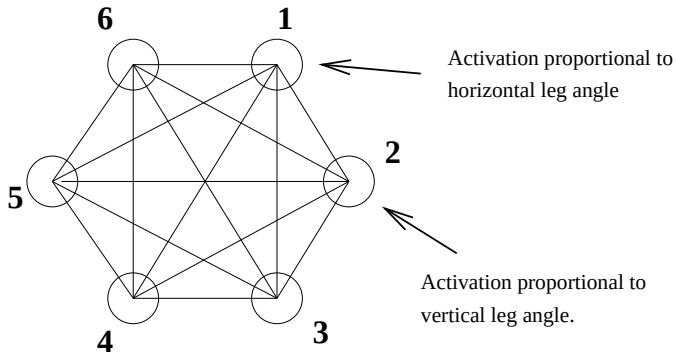

Figure 7.3: *Each leg controller consisted of six fully connected neurons. The activity of neuron 1 and neuron 2 controlled the horizontal and vertical leg angles respectively.*

Neural networks

While network parameters (connection weights, time constants, thresholds and so on) were under evolutionary control in the experiments described in this section, the overall shape of the network architecture was fixed to be the same for every member of the population. The repetitive movements characteristic of multi-legged walking behaviours were produced by a main oscillatory network of 8 coupled sub-networks, each responsible for the direct control of a single leg. The properties of this oscillatory network were then modulated by the output from three sensory neurons (one each for left and right infra red and one for the bumpers) and one permanently saturated bias neuron to produce the different movement patterns for walking forwards, backwards and turning. This architecture is very similar to, and was based upon, that used by Beer and Gallagher (1992). The components of this architecture will now be explained in detail.

Figure 7.3 shows one of the basic sub-networks responsible for the control of each leg. All eight sub-networks were identical in that only one set of sub-network parameters (threshold constants, connection weights and so on) was encoded on the genome and repeated eight times. These sub-networks consisted of six fully interconnected neurons, numbered 1 to 6 in the diagram, of the same type as those used by Beer and Gallagher (1992) and previously described in section 2.2.1. At each iteration, the input activity A_j of each of the $j = 1 \ to \ 6$ neurons in each of the 8 sub-networks was calculated according to equation 2.5

$$\tau_j \dot{A_j} = -A_j + \sum w_{ij} O_i + I_j$$

where τ_j was a time constant that affected the rate and extent to which the jth neuron responded to input, O_i was the output from the i_{th} neuron, w_{ij} was the weight on the connection from the i_{th} neuron to the j_{th} neuron, and I_j was any external input to the j_{th} neuron from outside the network. Once the input activity of each neuron had been calculated, the output O_j of each of the $j = 1 \ to \ 6$ neurons in each of the 8 sub-networks was calculated from the input activity A_j according to the sigmoid function of equation 2.2

$$O_j = (1 - e^{(t_j - A_j)})^{-1}$$

where t_j was a threshold constant associated with the j_{th} neuron. The range of possible values of each of these genetically specified constants is listed below.

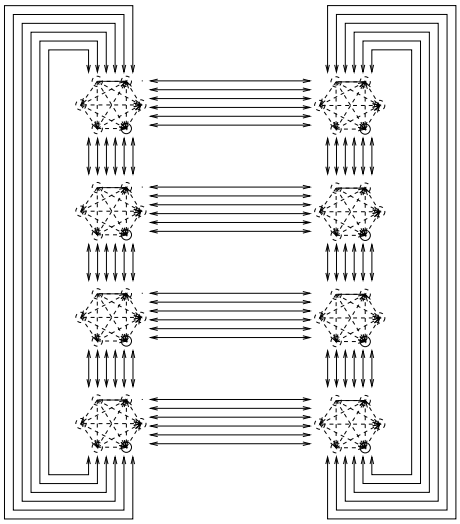

Figure 7.4: *This diagram shows how each leg-controller sub-network was coupled to the sub-network opposite it on the body and to the sub-network either side of it with wraparound.*

The output of neuron 1 in each sub-network was responsible for the signal to the servo-motor controlling the horizontal motion of the leg in question, and the output of neuron 2 was responsible for the signal to the servo-motor that controlled the vertical angle of the leg (see figure 7.3). For neuron 1, an output of 0 mapped onto a signal to the horizontal servo motor to point as far backwards as it could go, and an output of 1.0 mapped onto a signal to the the servo motor to point the leg as far forward as it could go. For neuron 2, outputs of 1 and 0 mapped onto signals to the vertical servo motor to position the leg in the fully up and down positions respectively.

Each sub-network was coupled to the sub-network directly opposite it and to the network on either side of it (with wraparound) as in figure 7.4. Each sub-network to sub-network coupling involved six symmetrical connections: from neuron 1 in one network to neuron 1 in the other, from neuron 2 to neuron 2, neuron 3 to neuron 3 and so on. All 4 cross-body couplings were identical in the sense that only six connection strengths were encoded on the genome and this set of six was repeated 4 times. All 8 along-body couplings were identical in the same way.

Figure 7.5 shows an example of how the connections between the neurons that made up the leg-controller sub-networks could be modulated by the sensor neurons and the bias neuron. Each connection between leg-controller neurons can be thought of as having had a synapse half way down its length that acted as a gate: open and the connection was unaffected, closed and the connection was switched off, effectively reducing the weight on the connection to zero. The synapse itself received input from sensor neurons and the bias neuron by way of weighted connections. If the total input to the synapse was greater than zero then the synapse gate was open and the connection between the leg-controller neurons was unaffected. If the total input to the synapse was less than zero then the synapse gate was closed and the weight on the connection between the leg-controller neurons dropped to zero.

Figure 7.6 shows how the three sensor neurons and the bias neuron were connected up to the synapses of the leg-controller sub-networks. Each of the thick black arrows represents 36 connections, one for each of the 36 synapses of a leg-controller sub-network. In total, three sets of

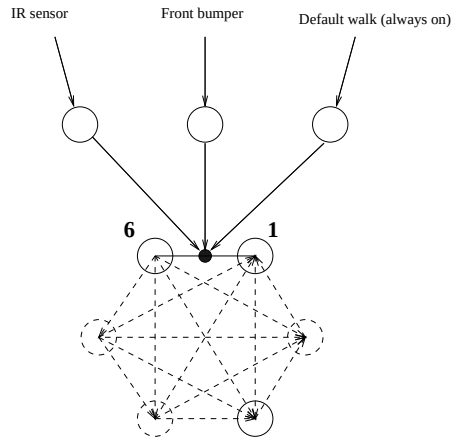

Figure 7.5: *This diagram shows how each connection between leg-controller neurons contains a synapse 'gate' that can be turned on or off by sensor neurons and the bias neuron. For the sake of diagrammatic simplicity only one connection is shown, whereas in reality every connection in every leg controller sub-network (*$36 \times 8 = 288$ *connections in total) contains a synapse that can be modulated in this way.*

36 connection weights were encoded on the genome: one set for the infra-red sensor neurons, one for the bumper sensor neuron and one for the bias neuron. Thus each of the two infra-red sensor neurons were connected to the synapses of the leg-controller sub-networks on the appropriate side by way of four identical sets of 36 connections (both sets of four were also identical to each other), and both the bumper sensor neuron and the bias neuron were connected up to all eight leg-controller sub-networks by way of eight identical sets of connections each.

A weighted input connection was associated with each of the three sensor neurons and the bias neuron. The signals from the infra-red sensors and bumper sensors that fed into these connections were normalised to lie within the range 0 to 1. In the case of the bias neuron, the signal that fed into its weighted input connection was permanently set at 1.

The network was updated iteratively using time-slicing techniques at a rate of 16 updates per second (or the simulated equivalent of a second). Also, in order to reduce computational overheads, a 200 place look-up-table was provided for the sigmoid function in place of the standard C-library maths functions.

Encoding scheme

Since the layout of the neural network architecture was fixed and predefined for every individual, a simple direct encoding scheme was employed. Every parameter was encoded on the genotype by a real number in the range 0 to 99, and this was mapped onto the relevant range during decoding. The parameters that were encoded and the ranges onto which they were mapped are as follows:

- 36 connection weights for the leg-controller sub-networks mapped onto the range ± 16.
- 12 cross-body and along-body coupling connection weights mapped onto the range ± 16.
- 36 infra-red sensor neuron to synapse connection weights mapped onto the range -6.5 to 25.5.
- 36 bumper sensor neuron to synapse connection weights mapped onto the range -6.5 to 25.5.

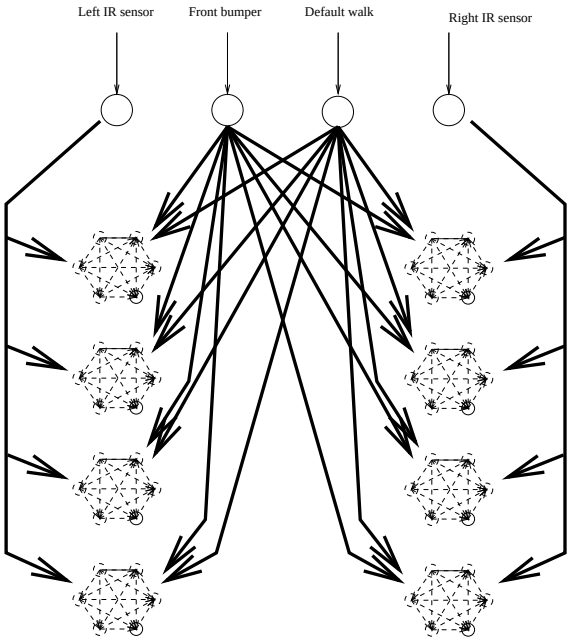

Figure 7.6: *This diagram shows how the sensory neurons and the always-on bias neuron were connected to the synapses of the leg controller sub-networks. Each of the thick black arrows represents 36 connections, one to each synapse in the leg controller. The cross-body and along-body couplings between leg-controller sub-networks have not been shown in this diagram.*

- 36 bias neuron to synapse connection weights mapped onto the range -6.5 to 25.5.
- 9 unit threshold constants mapped onto the range ± 4: 6 for the leg-controller sub-network neurons, 1 for the infra-red sensor neurons, 1 for the bumper sensor neuron and 1 for the bias neuron.
- 9 unit time constants mapped onto the range 0.5 to 5.0: 6 for the leg-controller sub-network neurons, 1 for the infra-red sensor neurons, 1 for the bumper sensor neuron and 1 for the bias neuron.
- 3 input connection weights mapped onto the range ± 16: 1 for the infra-red sensor neurons, 1 for the bumper sensor neuron and one for the bias neuron.

which makes a total of 177 parameters. Thus genotypes were strings of 177 numbers in the range 0 to 99.

Genetic algorithm and genetic operators

The genetic algorithm was an extremely simple generational model with tournament selection and elitism. After evaluating every member of the population, offspring genotypes were repeatedly produced until the next generation was full. To make a new offspring, two pairs of individuals were picked at random from the population and the fittest individuals from each pair (i.e. the winners of the tournaments) were chosen to act as parents. The offspring genotype was then formed from these two parents through a process of crossover and mutation: single point crossover was applied with a probability of 1, and every one of the 177 numbers that made up the offspring had a 0.02 chance of being mutated. A mutation involved changing the number in question by a random

amount taken from a roughly normal distribution with a standard deviation of around 18. If the new value was greater than 99 or less that 0 then it was clipped to lie within this range.

7.3 Experimental results

After removing some initial bugs from the code[1], reliably fit controllers evolved on practically every run within around 3500 generations. This took around 14 hours to run on a Sun Ultra SPARC and simulated over 11 weeks worth of real world evolution. When downloaded onto the real octopod, reliably fit controllers made the robot walk around its environment in a satisfactory manner, turning away from objects that fell within infra red range on both the right and the left hand side and backing away from objects that it hit with its bumpers.

Unfortunately, in this chapter we must make do with the bald statement of fact that evolved controllers successfully crossed the reality gap. In chapters 5, 6 and 8, demonstrations are provided of this fact, but this is not possible here due to both the nature of the octopod robot itself and the format of the thesis. If the robot was equipped with position sensors on each of the legs then data recorded from these sensors as the robot moved around a real-world environment could be used to provide such a demonstration. The robot, however, is not equipped with sensors of this type and data of the required type is not available. The other form such a demonstration could take, and probably the most natural, is the evidence provided by video footage of the robot wandering around its environment. This cannot, however, be profitably presented as part of a text and pictures document; even if a sequence of stills taken at short and regular time intervals were displayed, this would not be all that informative as to how the legs of the robot moved in the real world unless there were an impractically large number of them.

In lieu of any method of demonstrating how the legs of the real robot moved as it wandered around its environment, the best we can do is to provide a demonstration of how the motor signal patterns to these legs change in response to each of the four sensory scenarios. Figure 7.7 offers such a demonstration for a typical reliably fit controller that evolved after 3200 generations. From top to bottom, the first eight traces provide a novel representation of the motor signals issued to each leg over the course of an average fitness trial, and the bottom two traces show the resultant velocities of the left and right side of the simulated robot respectively. The best way of explaining how to read the slightly bizarre looking motor traces is to describe how they were generated. At each iteration of the simulation, a short line representing the current motor signal was added to the right hand side of each motor signal trace. As can be seen from the figure, these lines were of various thicknesses and were always drawn from the horizontal centre line of the trace either up and to the left or down and to the left with various different gradients. The thickness of each line represented the vertical angle of the leg relative to the ground as specified by the motor signal in question: the thicker the line, the lower the leg, and the thinner the line the higher the leg. The gradient of the line represented the horizontal angle of the leg relative to the body as specified by the motor signal in question: the further up and to the left, the further forwards relative to the body, and the further down and to the left, the further backwards relative to the body. In this

[1] One such bug, spotted by Jerome Kodjabachian, meant that the penalty due to robot instability was effectively applied at random. Surprisingly, even with such a fundamental error in the code, controllers evolved that were able to perform the task perfectly satisfactorily when downloaded onto the robot.

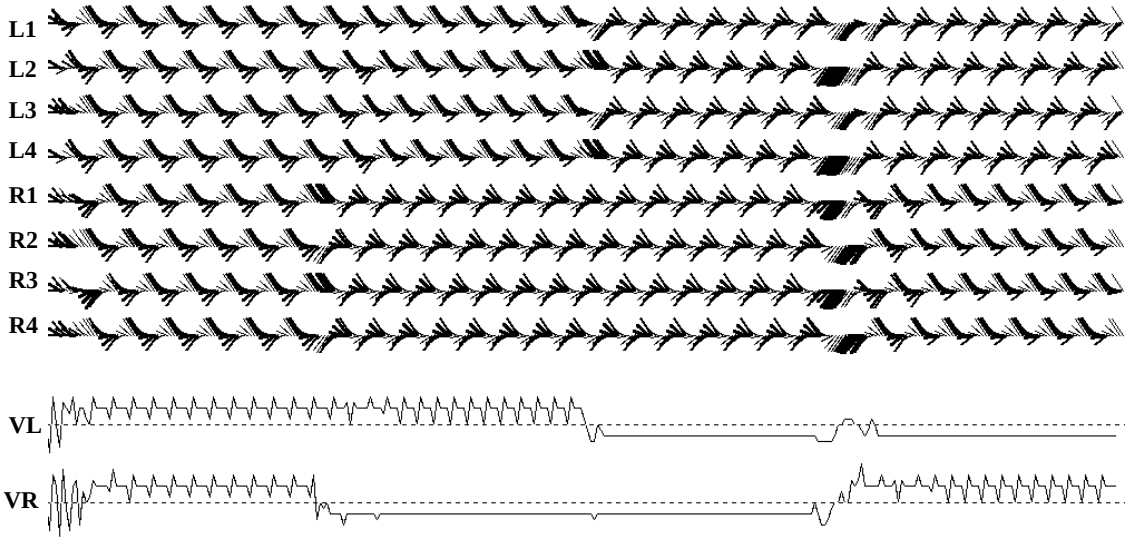

Figure 7.7: *Each leg controller consisted of six fully connected neurons. The activity of neuron 1 and neuron 2 controlled the horizontal and vertical leg angles respectively.*

way, although they are perhaps harder to read than other less informative types of trace devised to convey similar information (see (Beer and Gallagher 1992) for example), each of the motor signal traces of figure 7.7 represents *both* the vertical and horizontal components of the relevant signal over the course of a fitness test.

From the left and right velocity traces in figure 7.7 it is evident that the octopod moved forwards, then turned on the spot to the right, then backed up for a period and then rotated on the spot to the left. This corresponds to the order in which the four sensory scenarios arose during the fitness test that gave rise to this figure: no sensors active, left IR sensor active, bumpers and whiskers active, right IR active. Close inspection of the eight motor signal traces reveals:

- In the absence of any sensory activity the robot proceeded forwards using the classic tripod gate. Note that each leg is perfectly out of sync with the leg directly opposite it on the other side of the body.
- In response to activity from either of the two IR sensors, the motor signals sent to each of the legs on the side of the robot furthest from the sensor suddenly became the exact opposite of the signals sent to each of the corresponding legs on the side of the robot nearest the sensor. This made the side nearest the sensor signal go forwards and the one furthest away move backwards.
- In response to activity from the bumpers and whiskers, the robot proceeded backwards using a backwards tripod gate. Note that just before this phase of the simulation was finished, but well after the short-lived inputs to bumpers and whiskers had ceased, the robot paused with all legs down and back for a moment.

When downloaded onto the real robot, these motor patterns and walking gates were clearly and reliably recognizable.

7.4 Comments

The minimal simulation used in this chapter makes full use of the arguments put forward in section 3.3.2 to evolve controllers for the octopod robot. Simply put, these arguments state that a minimal simulation need only model the real-world dynamics involved in successful behaviour and no others. This is because the only controllers that must cross the reality gap, if the simulation is to be a success, are precisely those that use these dynamics (i.e. perform the behaviour) and no others. For many robotics setups and behaviours this may not be of any use since the dynamics involved in successful behaviour may be neither obvious ahead of time nor qualitatively different to the rest of the dynamics of the system. For the experiments reported in this chapter, however, the dynamics of the octopod robot during successful walking and obstacle avoiding behaviour were both relatively easy to identify and *much* easier to model than the dynamics of the octopod robot as a whole. A minimal simulation that modelled these dynamics alone was therefore easy to construct and ran extremely fast when compared to the simulation that would result from attempting to model *all* of the dynamics of the octopod robot within its environment.

Chapter 8

A minimal simulation for a complex sensor behaviour

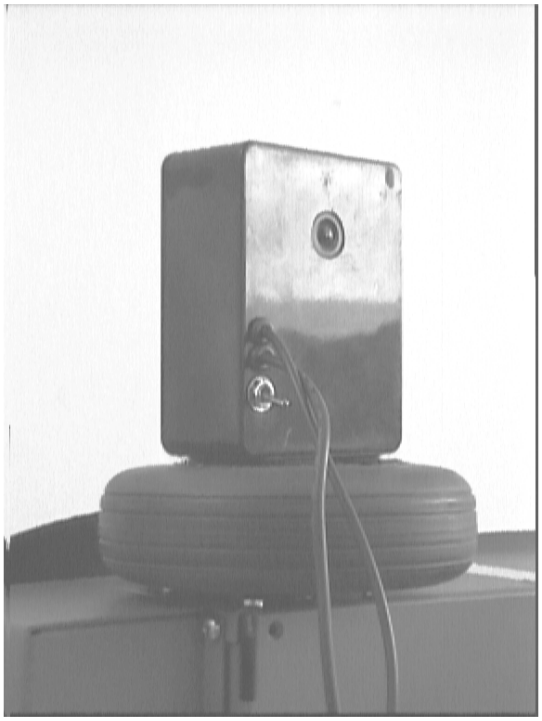

Figure 8.1: *The tracking camera head.*

In this chapter we report on experiments in which neural network controllers were evolved for a real-world robotics task that necessitated the non-trivial use of complex sensors. Figure 8.1 shows the robotic set-up used in these experiments. A small camera, connected to a frame-grabber, was mounted on a platform whose rotational velocity was under computer-control. The aim of the experiments was to evolve networks that would cause the camera to visually lock onto and track arbitrarily patterned rigid objects as they moved against an arbitrarily patterned static background.

Previous attempts to artificially evolve 'movement tracking' behaviours (Harvey, Husbands, and Cliff 1994; Miller and Cliff 1994; Floreano and Nolfi 1997) have allowed evolution to employ

some incidental property of the target object such as its colour or pattern to distinguish it from a background which has a different colour or pattern. Simple strategies can be very effective in these sorts of scenarios. For example, if the target object is black and the background is white then a strategy such as 'always turn towards the darkest part of the image' can be sufficient to track the object as it moves against its background. In the experiments reported here, however, the target object and background were both arbitrarily patterned. This means two things, firstly that evolving networks could only use the fact that the object was moving to pick it out from the background, and secondly that if the camera started to track the moving object, the camera's motion caused the background to move in the opposite direction within the image. Thus the evolutionary process not only had to locate and track movement caused by the object within the image, it also had to find some way of compensating for the movement within the image caused by the ego-motion of tracking itself. The mechanisms necessary for performing this sort of behaviour are far more complex than those needed for strategies like 'always turn towards the darkest part of the image'.

In order to make the problem tractable, only a sub-sampled region of the camera image was used in the experiments reported below. This region was a horizontal strip, three pixels wide and crossing the centre of the camera image, which was further sub-divided into 32 segments of equal length. The 32 average pixel values of these segments provided input to the controllers. Pre-processing the input in this way did not change the nature of the behaviour to be evolved, it just meant that controllers were evolved to perform motion-tracking using a 32 by 1 visual array as opposed to the 256 by 192 visual array provided by the raw camera image. However, preprocessing vastly reduced the computational overheads of both the controllers, which would otherwise have had to process many thousands of inputs, and the minimal simulation described below, which would otherwise have had to produce values for many thousands of inputs.

The minimal simulation used to evolve motion-tracking controllers is described in Section 8.1. The rest of the evolutionary machinery, including the neural networks, the encoding scheme, the genetic algorithm and genetic operators is described in section 8.2. Experimental results are put forward in section 8.3 and finally, in section 8.4, some comments are offered on the chapter as a whole.

8.1 The minimal simulation

A conventional approach to building a simulation for the evolution of visual behaviours (such as motion tracking) would most probably involve a model of how objects appear within the camera image that was as accurate as possible. The rationale being that if the simulated image was accurate enough, then from the point of view of evolving controllers, there would be no difference between simulation and reality. Accurately modelling how objects appear within a camera image, however, is an extremely computationally expensive and labour intensive activity. Even if the camera image is sub-sampled, as in the experiments reported here, it would still require ray-tracing algorithms, knowledge of the object's light-reflection properties and so on. If constructing a successful minimal simulation required accurately modelling how objects appeared in the camera image, evolving in the real world would probably be preferable.

Fortunately, the theory and methodology of minimal simulation put forward in chapter 3.3 does not advocate that the base set aspects of a minimal simulation should necessarily be as accurate as

possible. Instead, section 3.4.2 proposes that the base set aspects of a minimal simulation should be sufficiently varied, from trial to trial, so that controllers which evolve to be reliably fit are base set robust i.e. robust to the differences between the simulation and the real world. The minimal simulation described below depends fundamentally on this fact for its success. The model of how objects in the camera's field of view affect controller input is so simple as to be almost naive, yet the model is sufficiently varied from trial to trial that controllers which evolve to be reliably fit within the simulation satisfactorily track randomly patterned objects against similarly randomly patterned backgrounds in reality. Below, the step-by-step framework of section 3.5 is used to explain how the minimal simulation was constructed.

Precisely define the behaviour

The aim of the experiments was to evolve controllers that made the camera-head track randomly patterned objects of a range of sizes as they moved against a similarly randomly patterned background in a random fashion. This is a behaviour with many possible permutations of arbitrary properties, and to test enough of these to gain a reliable idea of a controller's fitness requires a fitness evaluation of many hundreds of trials. Because of this, steps were taken to reduce the amount of possible variation between trials. Within the simulation, evolving controllers were only tested on target objects that moved according to a particular pattern: across the field of view in a random direction, then stopping for a period, then moving again in either the same or a different direction. Although there was nothing special about this movement pattern in particular, it was complex enough that controllers which evolved to successfully track objects that moved in this fashion would also track objects that moved according to a large range of other movement patterns, and would, most probably, satisfactorily track objects that moved in a random fashion. It should be kept in mind, therefore, that in the experiments reported in this chapter, the behaviour that controllers were actually evolved to perform was to track objects that moved according to the specific movement pattern described above - not objects that moved arbitrarily. At any particular time, controllers were deemed to be successfully tracking an object if any part of the object fell within the field of view of the camera.

Identify the real-world base set

Since the criterion for behavioural success was concerned with the position of the object within the field of view of the camera, the base set consisted of all the features of the world that could affect this position. These included the way in which the target object moved relative to the background and the way in which the camera moved relative to the background in response to signals from the controller.

Build a model of the way in which the members of the base set interact with each other and react to controller output (when the robot is performing the behaviour)

In the simulation there was no model of the way in which the camera turned on the spot in order to track the target object, rotation was instead treated as a horizontal translation (as shown in figure 8.2). One variable held the horizontal position of the centre of the target, and another variable held the horizontal position of the centre of the camera's field of view. Rotation of the camera to the

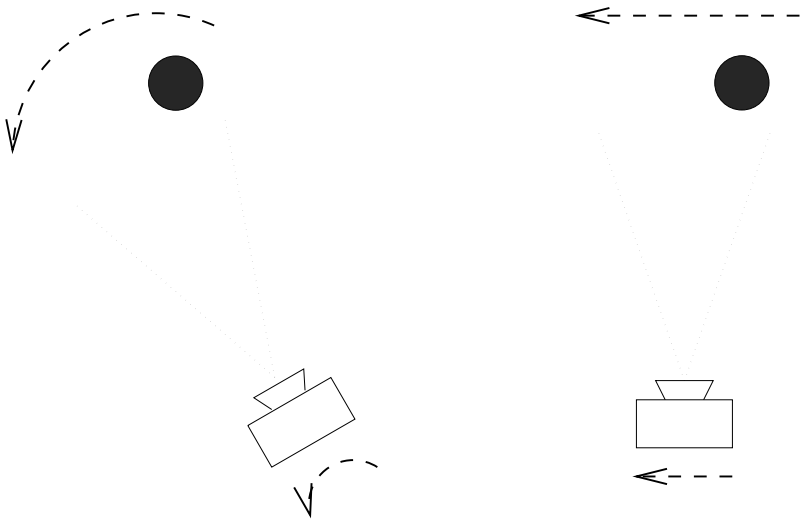

Figure 8.2: *This figure shows how a rotating camera tracking an object moving at a constant distance from the camera may be modelled by a translating camera that is tracking an object that is translating at a constant distance from the camera.*

right and left was modelled as a horizontal translation to the right and left. In order to track the target, therefore, the horizontal positions of the left and right edges of the camera's field of view had to contain the horizontal position of at least part of the target object, and the camera had to move horizontally in such a way that the target object never escaped these boundaries.

Build a model of (enough of) the way in which the members of the base set affect controller input (when the robot is performing the behaviour)

As explained above, a central horizontal strip of the camera image was divided up into 32 equal segments, and the 32 values formed by taking the average pixel value of each of these segments provided controllers with input. The way in which the background and target object affected the 32 values of the input array was modelled in an extremely simple fashion, and is shown diagrammatically in figure 8.3. At the start of each run an array of 10000 numbers between 0 and 255 (minimum and maximum grey-level pixel values) was initialized to act as the background. Also, at the start of every fitness trial, a small array of between 6 and 16 random numbers between 0 and 255 was initialized to act as the target object. At each iteration the horizontal position of the left-hand-side of the target object was used as an index into the background array and the target array was superimposed upon the background array at this point. The horizontal position of the left-hand-side of the camera image was then used as an index into the background array and the 32 following values were assigned as the 32 values of the input array. On each iteration, random deviates (generated by a simple multiplicative congruential random number generator (Press, Vetterling, Teukolsky, and Flannery 1992)) in the range ± 5 were added to each value in the image array. This corresponded roughly to the noise levels observed in reality.

Design a suitable fitness function

Each fitness trial consisted of the following three phases:

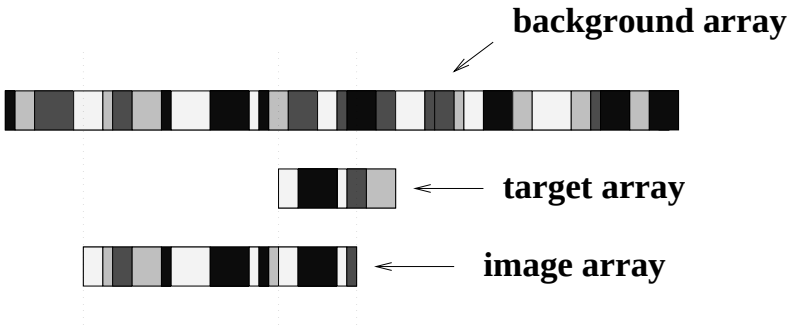

Figure 8.3: *This figure shows how the image array derived in the simulation from the target array superimposed on top of the background array.*

1. From a starting position just to the side of the field of view, an object moved towards (and across) the visual field for 10 seconds.
2. The object stopped moving completely for 5 seconds.
3. The object started moving again at the same speed as before but in a random direction. This phase also lasted 10 simulated seconds.

At each iteration δf was calculated from the difference between the horizontal position of the centre of the target array X_T and the horizontal position of the centre of the image array X_I according to the expression:

$$\delta f = \left\{ \begin{array}{ll} 0 & |X_I - X_T| > 16 \\ 16 - |X_I - X_T| & |X_I - X_T| \leq 16 \end{array} \right\}$$

At the end of each trial, the fitness function returned the final sum of the δfs. If the target object ever completely left the field of view at any point during a fitness trial then the trial was halted at that point, and the sum achieved thus far was returned as the fitness. Because of the inherently noisy nature of the task, each complete fitness evaluation returned the average value scored by an individual in a total of 40 different fitness trials. Each fitness trial was a maximum of 25 simulated seconds in length.

Ensure that evolving controllers are base set robust

Several parameters of the model of the base set were varied from trial to trial in order that reliably fit controllers were base set robust. Some of these parameters, such as the size of the target array, were varied widely so that controllers would evolve to track a whole range of different sizes of objects. Others, such as the momentum of the camera, were varied just enough to ensure that controllers would be robust to modelling inaccuracies. In particular:

- The size of the target array was varied between 6 and 16 input elements wide.
- The speed of the target object was randomly set between 0.5 and 1.5 input elements per simulation update.
- The initial direction of the target object, and whether, after the pause stage, it would continue in the same or the opposite direction was randomly determined at the start of each trial.

- The momentum of the camera was randomly set between 1 and 4.

In this way reliably fit controllers were forced to employ non-brittle strategies that could cope with objects of different sizes travelling at different speeds and in different directions. They were also forced to cope with cameras with different panning motor characteristics.

Ensure that evolving controllers are base set exclusive

The patterns of the target array and the background array were both treated as implementation aspects and varied from trial to trial in order that reliably fit controllers would be base set exclusive.

- The values that made up the target array were randomly set at the start of each trial in one of two different ways (picked at random): either they were all set to the same random value, or they were all set to different random values.
- The starting positions of the camera image and target object relative to the background array were randomly varied between 0 and 10000.

In this way, evolving controllers could only be reliably fit if they came up with a strategy that depended solely on movement in the image, and not on the shape or colour of the object or the background.

8.2 The evolutionary machinery

In this section we describe the evolutionary machinery that, together with the minimal simulation described above, was responsible for evolving neural network control architectures that could perform the behaviour satisfactorily in reality.

Neural networks

After trying several different types, the neural networks described below were found to be the most successful. However, in order to get this type of network to work satisfactorily, further preprocessing of the 32 input values was necessary. If the value of input i at time t was $I_i(t)$, then the corresponding value $J_i(t)$ made available to the neural networks was calculated according to the expression

$$J_i(t) = \left\{ \begin{array}{ll} 0 & |I_i(t) - I_i(t-1)| < 15 \\ 255 & |I_i(t) - I_i(t-1)| \geq 15 \end{array} \right\}$$

This meant that high network input could only be caused by sudden changes in the image, such as that resulting from image motion.

Apart from this extra preprocessing stage, the networks used in the experiments reported here were made from binary-valued neurons (either 'on' or 'off') with evolved weights and thresholds. In addition, a time constant was associated with each neuron that specified a period of time that the neuron would remain 'on' after the stimulus that caused it to turn 'on' in the first place had decayed. To update the state of the network at each iteration, the input activity A_j of each of the $j = 1$ *to* N neurons in the network was again calculated according to the simple weighted sum of equation

$$A_j = \sum O_i w_{ij} + J_j - A_{min}$$

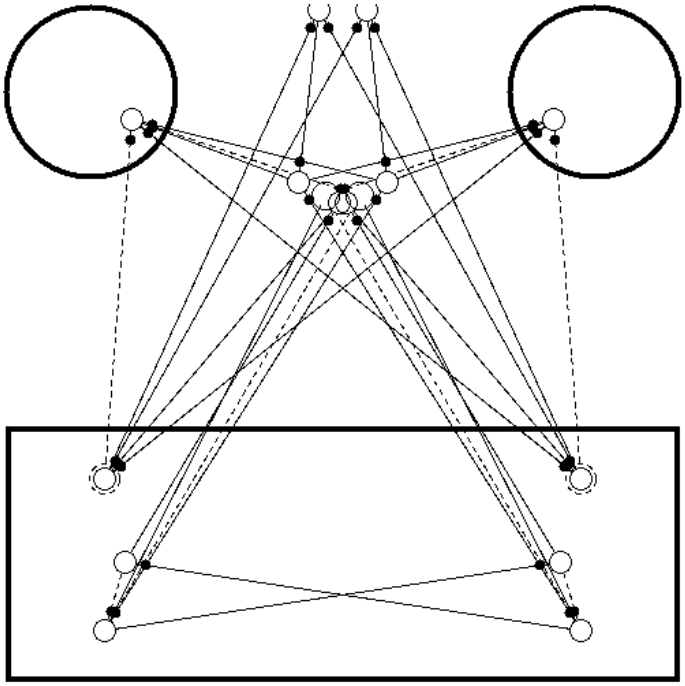

Figure 8.4: *The tracking network.*

where O_i was the output from the i_{th} neuron, w_{ij} was the weight on the connection from the i_{th} neuron to the j_{th} neuron, and J_j was any preprocessed external input to the j_{th} neuron. Each input activity A_j was then normalized to lie between 0 and 1 by reference to its maximum and minimum possible values. Using this normalized value $\bar{A}_j$ to calculate the output from each neuron increased the chance of the threshold lying within the dynamic range of the inputs, and thus increased the chance of random networks doing something rather than nothing (see section 2.2.1). After the normalized input activity of every neuron in the network had been calculated, the output O_j of all of the $j = 1\ to\ N$ neurons in the network was calculated according to a version of equation 2.6

$$O_j = \begin{cases} 0 & \bar{A}_j < t_j \\ 1 & \bar{A}_j \geq t_j \quad or \quad T_{\bar{A}_j \geq t_j} < \tau j \end{cases}$$

where $T_{\bar{A}_j \geq t_j}$ was the elapsed time since $\bar{A}_j \geq t_j$ was last true, τj was a time constant associated with the j_{th} neuron and t_j was a threshold constant associated with the j_{th} neuron.

Connection weights were in the range ± 2, thresholds were in the range 0 to 1, and time delays were in the range 0 to 16 network updates. The state of the network was updated at the rate of 32 times a second (or the simulated equivalent of a second): twice for each update of the inputs and outputs which were undertaken at the rate of 16 times a second (or the simulated equivalent of a second).

The encoding scheme

The encoding scheme was again a version of the spatially determined encoding scheme described in section 2.3. This type of scheme, although simple, allows genotypes to grow under genetic control with a minimum amount of phenotypic disruption.

Figure 8.4 shows an evolved network and the two dimensional space in which development took place. The location of each neuron within the space was genetically specified and determined its function within the network: a location within either of the two circular regions at the top of the developmental space meant a motor neuron, a location within the large rectangular region at the bottom of the developmental space meant an input neuron, and a location anywhere else meant a hidden neuron. The output signal to the motors was calculated according to the relation $signal = k \times (O_l - O_r)$, where O_l and O_r were the average output values of the neurons in the left and right motor regions respectively and k was a constant equivalent to around 180° per second. As explained in section 8.1 above, there were a total of 32 inputs made available to evolving networks, each corresponding to a certain number of pixels situated on the horizontal line drawn through the middle of the camera image. To determine which of these 32 inputs each input neuron corresponded to, the rectangular input region of the developmental space (see figure 8.4) was itself divided up into 32 rectangular input sub-regions running horizontally from left to right, each corresponding (in order) to one of the 32 inputs. Input values were normalised to lie within the range 0 to 1. Inputs and outputs were updated at a speed of 16 times per second (or the simulated equivalent of a second).

Genotypes consisted of a variable number of genes, each encoding the necessary parameters for a neuron. In fact, since bilateral symmetry was imposed on evolving networks by reflecting each neuron across the midline of the developmental space, each gene actually coded the parameters for not one but two neurons. Each gene was 17 integers long, and each integer lay between 0 and 99. The first number of each gene stored the gene's 'age' for mutation locking purposes and played no role in determining the structure of the phenotype (see section 2.4.2 and the explanation given below of the genetic operators used). The next two numbers of each gene specified the x and y coordinates of the corresponding neuron's position within the developmental space. The fourth number specified the neuron's threshold, and the fifth number specified the neuron's decay constant. The next six numbers specified the target positions and weights for two connections *to* other neurons *from* the neuron in question, and the next two numbers specified the target positions and weights for two connections *from* other neurons *to* the neuron in question.

Genetic algorithm and genetic operators

The genetic algorithm was a simple generational model with truncation selection and elitism. Mutation locking (see section 2.4.2) was also implemented. After evaluating every member of the population, the next generation was formed by asexually producing the required number of offspring. At each offspring event, a parent genotype was selected at random from the fittest 50% of the population and subjected to a number of different forms of mutation to produce an offspring. First of all, every unlocked gene on the genotype had a 0.05 chance of undergoing a major 'innovative' mutation which involved being completely randomized and having its age reset to zero. Secondly, every parameter on the genotype, *including* those of genes that were mutation locked had a small chance of undergoing a 'tuning' mutation: the adding of a random offset in the range $\pm 5\%$. The number of tuning mutations per genotype was picked from a Poisson distribution with an expected number of 5. Finally, there was a chance of 0.01 of both adding a completely random gene aged zero or deleting a randomly selected unlocked gene entirely. After enough

Figure 8.5: *A view of the gantry robot taken directly from the panning camera head. The gantry, visible on the left of the image, has had a large dark cylinder of paper with vertical white wavy lines wrapped around it.*

Figure 8.6: *A view of the gantry robot taken directly from the panning camera head. The gantry, visible on the right of the image, has had a small white cylinder of paper attached to the side of the gantry facing the camera.*

offspring had been generated, the age of every gene of every individual in the population was increased by 1. Those genes that passed the age of 200 were regarded as mutation-locked and made immune from both innovative mutations and deletion.

8.3 Experimental results

Out of a total of five runs with the parameter settings given above, only one run evolved networks that could reliably track objects within the simulation, consistently achieving near-optimal fitness on every fitness trial. Of the other four runs, two evolved networks that performed reasonably satisfactory motion-tracking within the simulation, achieving an average of around 80% of maximum fitness on each complete fitness evaluation, and the other two evolved networks that were only capable of reliably achieving around 50% of maximum fitness.

Figure 8.4 shows one of the networks that evolved to reliably perform motion-tracking behavior within the simulation. The 1300 generations that this network took to evolve required around 24 hours to run on a Sun SPARC Ultra and simulated over 5 years' worth of real world evolution. The network employs a saccading strategy (Carpenter 1977) that tracks moving objects through a series of jumps and pauses instead of moving smoothly and at the same speed as the object. Furthermore, the strategy is directionally sensitive in that the network will only saccade to track an object if it is moving away *from* the centre of the image *towards* the image boundary, and not if it is moving *towards* the centre of the image *from* the image boundary. This directional sensitivity means that the simulated camera only saccades in the direction of motion of the object.

In order to provide some objective evidence of its ability to transfer across the reality gap, the network was downloaded onto the panning camera head set-up described at the beginning of this chapter and subjected to a specially devised real-world tracking task involving the gantry robot described in chapter 6. For this task, one of the long walls of the gantry arena was removed and the remaining three walls were decorated in as random-looking a fashion as possible with pieces of black, white and grey card. The head of the gantry robot was also decorated, and a short

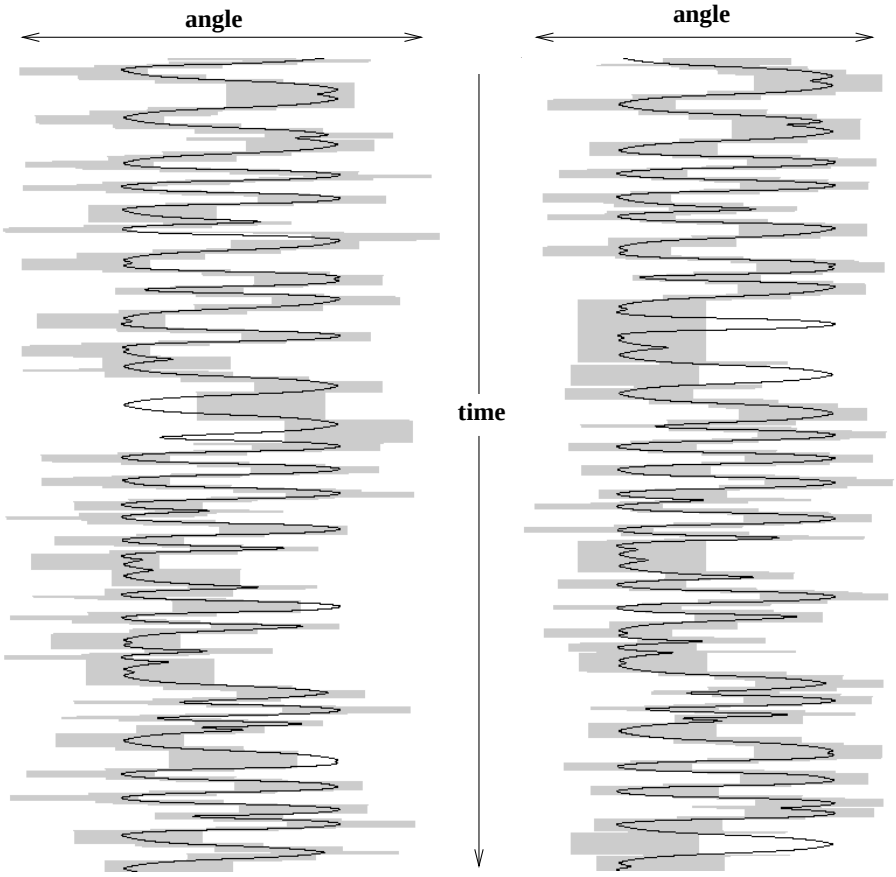

Figure 8.7: *Each of these two traces derive from a continuous fifteen minute test of the ability of the neural network of figure 8.4 to track moving objects in the real world. The left-hand trace resulted from testing on the large dark object pattern of figure 8.5 and the right-hand trace resulted from testing on the small white object pattern of figure 8.6. The thick light grey line in each of these traces plots the angular position of the panning camera head's field of view over time. The thin black line plots the relative angular position of the centre of the moving object (i.e. the gantry robot) over time.*

computer program was written that made it parade up and down the arena, parallel to the missing wall, and changing speed and direction at random. The panning camera head was then positioned half way down the length of the missing wall, just outside the arena and facing inwards. The task was to track the randomly patterned object, provided by the gantry robot, as it moved against the randomly patterned background, provided by the remaining three walls of the gantry arena.

Figures 8.5 and 8.6 show images (taken directly from the panning camera head) of the gantry robot decorated as two different patterned objects and viewed against the randomly patterned background provided by the walls of the arena. The network was tested continuously for fifteen minutes on each of these two object patterns and the angular position of both the panning camera head and the gantry robot relative to the panning camera head were automatically and continuously recorded during each test. Figure 8.7 shows two traces, one for each fifteen minute test, of these angular positions plotted over time. The thick light grey line in each of these traces plots the angular position of the panning camera head's field of view over time. The thin black line plots the relative angular position of the centre of the moving object (i.e. the gantry robot) over time. In total, the network kept the large dark object pattern of figure 8.5 within the field of view of the

tracking camera for 91% of the fifteen minute test period. It kept the small white object pattern of figure 8.6 within the field of view of the tracking camera for 92% of the fifteen minute test period.

8.4 Comments

The minimal simulation used in these experiments relied heavily on the fact that reliably fit controllers, evolved in a simulation with very inaccurate base set aspects and lots of implementation aspects, are just as likely as any others to cross the reality gap *provided* that the right amount of random variation is included in the simulation in the right way according to the methodology laid out in section 3.4. This meant that a fast-running, easy to implement model of how objects affected controller input could be used to successfully evolve real-world tracking behaviours even though it was inaccurate to the point of naivety. The savings in both construction time and running time due to the simplicity of this model are both especially apparent when one considers the computationally intensive ray-tracing algorithms, empirically derived reflection coefficients and so on that are part and parcel of more conventional visual simulations.

The novel neural networks used in these experiments seemed to work well, but only after an extra preprocessing stage had been implemented. This extra stage meant that neurons responded to sudden changes in the values of the 32 pixel fields rather than the values themselves. With all else being equal, therefore, neuronal input was a direct consequence of image motion. The experimental results should be viewed with this in mind, since this preprocessing stage greatly simplified one crucial aspect of the motion-tracking task: that of reacting to image motion in the first place. However, this should not detract from the fact that the evolved motion-tracing networks performed a complicated real-world behaviour. Even given image motion for free, they discriminated image motion due to the target object from image motion due to the movement of the camera, executed well-timed saccades to track the target object, and executed these saccades only in the direction of movement of the target object to minimize the risk of losing it from the field of view.

Chapter 9

Conclusions

The fact that this thesis tackles one of the most pressing challenges for Evolutionary Robotics at the moment opens the door for research into how to tackle many of the others. At the time of writing, in the EASY group at Sussex University and elsewhere, researchers are using minimal simulations to investigate various issues including: the evolution of diffusing gas networks for robot control (Husbands 1998), the effects of noise on the evolution of complexity (Seth 1998), robot football (Smith 1998), encoding schemes for evolving repeated structure, and even whether evolution is the best search strategy for automatically designing robot controllers in the first place. The point is that all of these people are doing real Evolutionary Robotics research into non-trivial problems without having to work with real robots, and without having to spend too much time actually evolving. In that it allows researchers to focus their attention on issues that would otherwise not be practical to explore, this thesis pushes Evolutionary Robotics one step further towards seriously competing with more traditional techniques for the design of robot controllers. Below the major contributions of the thesis are examined in turn and directions for future work are highlighted in each case.

The arguments of chapter 3 that led finally to a step by step guide to building a minimal simulation for Evolutionary Robotics proceeded in three stages:

1. The inevitable differences between simulation and reality were examined, and the reasons why some controllers can overcome these differences to successfully transfer into reality while others cannot were identified. This lead to the development of two conditions that controllers must fulfill if they are to cross from simulation into reality.

2. In order for it to be possible that controllers can fulfill these conditions, a simulation must model certain real-world features and processes, and these were identified. A simulation that models no more than the minimum necessary was labelled a minimal simulation.

3. By no means every controller evolved in a minimal simulation will fulfill the conditions for successful transfer, and techniques were proposed for using the evolutionary process itself to force evolving controllers to meet them. Controllers are not only evolved to perform a specific behaviour within a minimal simulation, therefore, they are also evolved to fulfill the conditions for successful transfer into reality.

One very important point to notice in terms of directions for future work is that only the last of

these three stages actually refers to the evolutionary process itself, and although the minimal simulation approach was developed for use in Evolutionary Robotics, there are many other branches of adaptive robotics that could also benefit from fast-running easy-to-build simulations. The first stage identifies conditions that controllers must fulfill if they are to cross into reality and the second stage identifies the minimal simulations in which such conditions can be met: both of which remain true no matter how controllers come about. Only the third stage refers to the evolutionary process (by proposing techniques for forcing controllers that evolve to perform a particular behaviour to also fulfill the conditions for successful transfer), and we can imagine similar third stages for adaptive processes other than evolution. For example, very similar techniques might be used to force controllers which *learn* to perform a particular behaviour over a number of trials within a minimal simulation to also fulfill the conditions for successful transfer. In this way, minimal simulations may turn out to have far-reaching consequences for the adaptive robotics community as a whole, and not just for those interested in evolutionary approaches.

Chapter 4 presented a formal treatment of the theory behind minimal simulations. It introduced a logical formalism for reasoning about controllers performing behaviours in environments and *derived* a minimal set of conditions for successfully crossing the reality gap from the same set of assumptions as those made in chapter 3. The fact that these conditions corresponded closely to those put forward in chapter 3 provides good evidence for the logical soundness of the arguments underlying the minimal simulation approach. The fact that they did not correspond exactly, however, highlights the fact that it is never possible to be 100% certain that an evolved controller will successfully transfer from a minimal simulation into reality, only extremely confident. In practice, it was argued, this is all we need. The formalism introduced in this chapter should be seen as part of the ongoing research effort being undertaken by several researchers to find general techniques for reasoning in principled ways about agents performing behaviours in environments (Smithers 1994; Beer 1995a; Pfeifer and Scheier 1998). Future successes in this area may help us to both understand existing robot behaviours and design new ones.

Using minimal simulations built according to the methodology of chapter 3, in conjunction with many of the techniques described in chapter 2, controllers were evolved that could perform the following behaviours:

- **T-maze solving behaviour for a Khepera robot.** A T-maze environment was constructed in which a beam of light could be shone across the the first corridor from either side. Controllers were evolved to guide a Khepera robot through the T-maze, 'remembering' from which side the beam of light was shone and turning down the corresponding corridor arm at the junction.
- **Shape-discrimination behaviour for the gantry robot.** An equilateral triangle and a square of white paper were both stuck onto a long wall of an otherwise black arena. Starting from different positions and orientations, controllers were evolved to steer the gantry robot towards the triangle while ignoring the square.
- **Walking and obstacle-avoiding behaviour for an octopod robot.** Controllers were evolved to make the octopod robot walk around its environment, turning away from objects that fall within range of the IR sensors and backing away from objects that touch the front bumpers and whiskers.
- **Motion-tracking behaviour for a panning camera-head.** Controllers were evolved to

make a simple panning camera-head track arbitrarily patterned objects as they moved against arbitrarily patterned backgrounds.

In each case, controllers were evolved in a matter of hours using minimal simulations that would otherwise have taken months or even years to evolve in reality; they also successfully transfered onto the real robots.

In (Mataric and Cliff), the authors suggest that as robots and the behaviours we want to evolve for them become more and more complex, simulations will become either so computationally expensive that all speed advantages over real-world evolution will be lost, or so hard to design that the time taken in development will outweigh the time saved in reality. This thesis has demonstrated that for certain types of behaviours and robots, at least, this will *not* be the case. The experiments of chapters 6 and chapter 8, for instance, show that it is possible to create minimal simulations for robots which employ complex sensory modalities such as vision, and the experiments of chapter 7 show that it is possible to create minimal simulations for robots that require complex motor coordination. The experiments of chapter 5, while not involving the evolution of behaviours that are particularly complicated in themselves, show that it is possible to create minimal simulations for the evolution of behaviours that are. To illustrate this, consider a slightly extended version of the minimal simulation used in these experiments in which the robot is not just presented with a single junction at the end of the first corridor but a whole series of junctions that together add up to a complex maze. In the first corridor, furthermore, the robot does not just pass a single light signal but a whole series, placed one after another, some on the left and some on the right, that together signal the correct path through the maze that follows. This is an extremely complex behaviour to evolve by today's standards of what can and cannot be evolved, and yet the necessary minimal simulation remains simple and fast.

The point is that whether a minimal simulation is easy to construct and runs fast depends not on the complexity of the behaviour we want to evolve using it, nor on the complexity of the robot that it simulates, but only on the complexity of the base set of environmental features necessary to underly the behaviour for that robot. Provided these are simple enough, then the behaviour and/or robot can be arbitrarily complex. It is too early to say much about the complexity of the robot-environment interactions employed by the robots and control architectures of the future, but consider two points. Firstly, results in insect and invertebrate neuroscience suggest that many complex behaviours are often accomplished by way of simple interactions with the environment rather than complicated ones (Collett 1996; Wehner 1987; Horridge 1992). And secondly, control strategies grounded in complex robot-environment interactions can lead to prohibitively heavy real-time processing requirements (Brooks 1991b): a fact that has fuelled the trend in mobile robotics over the last few years from the internal world model making robots of the seventies (Nilsson 1984) to the current low level behaviour based robotics of the present day (Chiel, Beer, Quinn, and Espenschied 1992). Whether the Minimal Simulation approach will scale up, therefore, remains to be seen.

Bibliography

Aizawa, A. and B. Wah (1994). Scheduling of genetic algorithms in a noisy environment. *Evolutionary Computation 2*(2), 239–266.

Back, T., F. Hoffmeister, and H. Schwefel (1991). A survey of evolution strategies. In R. Belew and L. Booker (Eds.), *Proceedings of the 4th International Conference on Genetic Algorithms*, pp. 2–9. Morgan Kaufmann.

Back, T. and H. Schwefel (1993). An overview of evolutionary algorithms for parameter optimisation. *Evolutionary Computation 1*(1), 1–23.

Barhen, J., W. Dress, and C. Jorgensen (1987). Applications of concurrent neuromorphic algorithms for autonomous robots. In R. Eckmiller and C. Malsburg (Eds.), *Neural Computers*, pp. 321–333. Springer-Verlag.

Beer, R. (1990). *Intelligence as Adaptive Behavior: An Experiment in Computational Neuroethology*. San Diego, California: Academic Press.

Beer, R. (1995a). A dynamical systems perspective on agent environment interaction. *Artificial Intelligence 72*(1-2), 173–215.

Beer, R. (1995b). On the dynamics of small continuous time neural networks. *Adaptive Behaviour 3*(4), 469–509.

Beer, R. (1996). Toward the evolution of dynamical neural networks for minimally cognitive behaviour. In P. Maes, M. Mataric, J.-A. Meyer, J. Pollack, and S. W. Wilson (Eds.), *Proceedings of the fourth international conference on simulation of adaptive behaviour*, Cambridge, Mass, pp. 421–429. MIT Press.

Beer, R. and J. Gallagher (1992). Evolving dynamic neural networks for adaptive behavior. *Adaptive Behavior 1*, 91–122.

Braitenberg, V. (1984). *Vehicles: Experiments in Synthetic Psychology*. Cambridge MA: MIT Press/Bradford Books.

Brogan, W. L. (1991). *Modern control theory* (3rd ed.). Prentice-Hall.

Brooks, R. (1991a). Elephants don't play chess. In P. Maes (Ed.), *Designing Autonomous Agents: Theory and Practice from Biology to engineering and Back*, pp. 3–15. Cambridge: massachusetts: M.I.T. Press.

Brooks, R. (1991b). Intelligence without reason. In *Proceedings of the Twelth International Joint Conference on Artificial Intelligence*, San Mateo, California, pp. 569–95. Morgan Kauffman.

Brooks, R. (1991c). Intelligence without representation. *Artificial Intelligence 47*, 139–159.

Brooks, R. (1992). Artificial life and real robots. In F. J. Varela and P. Bourgine (Eds.), *Toward a Practice of Autonomous Systems: Proceedings of the first European Conference on Artificial Life*, Cambridge, Massachusetts, pp. 3–10. MIT Press / Bradford Books.

Calabretta, R., R. Galbiati, S. Nolfi, and D. Parisi (1996). Two is better than one: A diploid genotype for neural networks. *Neural Processing Letters 4*(3), 149–155.

Carpenter (1977). *Movements of the Eyes*. London: Pion Press.

Chiel, H., R. Beer, R. Quinn, and K. Espenschied (1992). Robustness of a distributed neural network controller for locomotion in a hexapod robot. *IEEE Transactions on Robotics and Automation 8*(3), 293–303.

Cliff, D. (1991). Computational neuroethology: a provisional manifesto. In J.-A. Meyer and S. W. Wilson (Eds.), *Proceedings of the First International Conference on Simulation of Adaptive Behavior*, Cambridge, Massachusetts. M.I.T. Press / Bradford Books.

Cliff, D. (1994). Ncage. neural control architecture genetic encoding. Cognitive Science Research Paper 325, School of COGS, University of Sussex.

Cliff, D., I. Harvey, and P. Husbands (1993). Explorations in evolutionary robotics. *Adaptive Behavior 2*, 73–110.

Cliff, D., P. Husbands, and I. Harvey (1993a). Analysis of evolved sensory motor controllers. In *Proceedings of the Second European Conference on Artificial Life*, pp. 192–204.

Cliff, D., P. Husbands, and I. Harvey (1993b). Evolving viually guided robots. In H. R. J. Meyer and S. Wilson (Eds.), *Proceedings of the 2nd International conference on the Simulation of Adaptive Behaviour*, pp. 374–383. MIT Press/Bradford Books.

Collett, T. S. (1996). Insect navigation en-route to the goal - multiple strategies for the use of landmarks. *Journal of Experimental Biology 199*(1), 227–235.

Collins, R. and D. Jefferson (1991). Selection in massively parallel genetic algorithms. In R. K. Belew and L. B. Booker (Eds.), *Proceedings of the Fourth Intl. Conf. on Genetic Algorithms, ICGA-91*, pp. 249–256. Morgan Kaufmann.

Colombetti, M. and M. Dorigo (1992). Learning to control an autonomous robot by distributed genetic algorithms. In J.-A. Meyer, H. Roitblat, and S. Wilson (Eds.), *Proc. of 2nd Intl. Conf. on Simulation of Adaptive Behavior, SAB'92*, pp. 305–312. MIT Press/Bradford Books.

De Jong, K. (1975). *Analysis of the Behavior of a Class of Genetic Adaptive Systems*. Ph. D. thesis, Univ. of Michigan.

Dellaert, F. and R. Beer (1996). A developmental model for the evolution of complete autonomous agents. In P. Maes, M. Mataric, J.-A. Meyer, J. Pollack, and S. Wilson (Eds.), *From Animals to Animats IV*, Cambridge, MA, pp. 393–402. MIT Press.

Dellaert, F. and R. D. Beer (1994). Co-evolving body and brain in autonomous agents using a developmental model. Technical Report CES-94-16, Case Western Reserve University.

Dorigo, M. and M. Colombetti (1997). *Robot Shaping: an Experiment in Behaviour Engineering*. MIT Press/Bradford Books.

Dorigo, M. and U. Schnepf (1993). Genetic-based machine learning and behavior-based robotics: A new synthesis. *IEEE Transactions on Systems, Man, Cybernetics 23*(1), 141–154.

Eigen, M. (1987). New concepts for dealing with the evolution of nucleic acids. *Cold Spring Harbour Symposia on Quantitative Biology L II*.

Fitzpatrick, J. and J. Grefenstette (1988). Genetic algorithms in noisy environments. *Machine Learning 3*(2), 101–120.

Floreano, D. and F. Mondada (1994). Automatic creation of an autonomous agent: Genetic evolution of a neural -network driven robot. In D. Cliff, P. Husbands, J.-A. Meyer, and S. Wilson (Eds.), *Proceedings of the Third International Conference on the Simulation of Adaptive Behavior*, Volume 3, pp. 421–431. MIT Press/Bradford Books.

Floreano, D. and F. Mondada (1996a). Evolution of homing navigation in a real mobile robot. *IEEE Transactions on Systems, Man and Cybernetics–Part B: Cybernetics 26*(3), 396–407.

Floreano, D. and F. Mondada (1996b). Evolution of plastic neurocontrollers for situated agents. In P. Maes, M. Mataric, J.-A. Meyer, J. Pollack, and S. Wilson (Eds.), *From Animals to Animats IV*, Cambridge, MA, pp. 402–411.

Floreano, D. and S. Nolfi (1997). Adaptive behaviour in competetive co-evolutionary robotics. In P. Husbands and I. Harvey (Eds.), *Proceedings of the 4th European Conference on Artificial Life*, Cambridge, Mass, pp. 378–387. MIT Press.

Fukuda, T. (1989). Structure decision method for self organizing robots based on cell structure-cebot. In *Proceedings of International Conference on Robotics and Automation*.

Gallagher, J., R. Beer, K. Espenschied, and R. Quinn (1996). Application of evolved locomotion controllers to a hexapod robot. *Robotics and Autonomous Systems 19*, 95–103.

Gallagher, J. C. and R. D. Beer (1993). A qualitative analysis of evolved locomotion controllers. In J.-A. Meyer, H. Roitblat, and S. Wilson (Eds.), *Proceedings of the Second International Conference on Simulation of Adaptive Behaviour (SAB92)*, pp. 71–80. Cambridge, MA: MIT Press/Bradford Books.

Garis, H. D. (1991). Genetic programming: Artificial nervous systems, artificial embryos and embryological electronics. In H. Schwefel (Ed.), *Parallel Problem Solving from Nature*. Springer-Verlag.

Goldberg, D. (1989a). Sizing populations for serial and parallel genetic algorithms. In D. Schaffer (Ed.), *Genetic Algorithms and their Applications: Proceedings of the Third International Conference on Genetic Algorithms*, pp. 70–79. Morgan Kaufmann.

Goldberg, D. E. (1989b). *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, Massachusetts: Addison-Wesley.

Gruau, F. (1992). Cellular encoding of genetic neural networks. Technical Report 92-21, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon.

Gruau, F. (1994). *Neural Network Synthesis using Cellular Encdoing and the Genetic Algorithm*. Ph. D. thesis, Ecole Normale Superieure de Lyon.

Gruau, F. (1995). Automatic definition of modular neural networks. *Adaptive Behavior 3*(2), 151–184.

Gruau, F. (1997). Cellular encoding for interactive evolutionary robotics. In P. Husbands and I. Harvey (Eds.), *Proceedings of the Fourth European COnference on Artificial Life*, Cambridge, MA, pp. 368–377. MIT Press.

Harvey, I. (1992). Species adaptation genetic algorithms: the basis for a continuing saga. In F. J. Varela and P. Bourgine (Eds.), *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, Cambridge, Massachusetts, pp. 346–354. M.I.T. Press / Bradford Books.

Harvey, I. and P. Husbands (1992). Evolutionary robotics. In *Proceedings of IEE Colloquium on 'Genetic Algorithms for Control Systems Engineering', London 8 May 1992*.

Harvey, I., P. Husbands, and D. Cliff (1994). Seeing the light: Artificial evolution, real vision. In D. Cliff, P. Husbands, J.-A. Meyer, and S. Wilson (Eds.), *Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, Volume 3, pp. 392–402. MIT Press/Bradford Books.

Higuchi, T., T. Niwa, T. Tanaka, H. Iba, H. D. Garis, and T. Furuya (1992). Evolving hardware with genetic learning: A first step towards building a darwin machine. In J.-A. Meyer, H. Roitblat, and S. Wilson (Eds.), *Proc. of 2nd Intl. Conf. on Simulation of Adaptive Behavior, SAB'92*, pp. 417–424. MIT Press/Bradford Books.

Hinton, G., J. McClelland, and D. Rumelhart (1986). Distributed representations. In D. Rumelhart, J. McClelland, and the PDP Research Group (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, pp. 77–109. MIT Press.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor, USA: University of Michigan Press.

Holland, J. (1987). Genetic algorithms and classifier systems: Foundations and future directions. In J. J. Grefenstette (Ed.), *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, Hillsdale, N.J., pp. 82–89. Lawrence Erlbaum Associates.

Hopfield, J. J. (1984). Neurons with graded response properties have collective response properties like those of two-state neurons. In *Proceedings of the National Academyof Sciences of the United States of America*, Number 81, pp. 3088–3092.

Horridge, G. (1992, September). What can engineers learn from insect vision. *Philosophical Transactions of the Royal Society of London 337*(1281), 271–282.

Husbands, P. (1997). Personal communication.

Husbands, P. (1998). Evolving robot behaviours with diffusing gas networks. In P. Husbands and J.-A. Meyer (Eds.), *Proceedings of Evorob98*. Springer Verlag.

Husbands, P. and I. Harvey (1992). Evolution versus design: Controlling autonomous robots. In *Integrating Perception, Planning and Action: Proceedings of the Third Annual Conference on Artificial Intelligence, Simulation and Planning*, pp. 139–146. IEEE Press.

Husbands, P., I. Harvey, and D. Cliff (1993a). Analysing recurrent dynamical networks evolved for robot control. In *Proc. 3rd IEE Int. Conf. on ANNs*, pp. 158–162. IEE Press.

Husbands, P., I. Harvey, and D. Cliff (1993b). An evolutionary approach to situated a.i. In A. Sloman, D. Hogg, G. Humphreys, A. Ramsay, and D. Partridge (Eds.), *Prospects for Artificial Intelligence*. I.O.S. Press.

Husbands, P., I. Harvey, and D. Cliff (1995). Circle in the round: state space attractors for evolved sighted robots. *Robotics and Autonomous Systems 15*(1-2), 83–106.

Husbands, P., I. Harvey, D. Cliff, and G. Miller (1994). The use of genetic algorithms for the development of sensorimotor control systems. In P. Gaussier and J.-D. Nicoud (Eds.), *Proceedings of From Perception to Action Conference*, pp. 110–121. IEEE Computer Society Press.

Husbands, P., I. Harvey, D. Cliff, and G. Miller (1997). Artificial evolution: A new path for ai? *Brain and Cognition* (34), 130–159.

Husbands, P., I. Harvey, N. Jakobi, A. Thompson, and D. Cliff (1997). A case study in evolutionary robotics. In T. Back, D. Fogel, and Z. Michalewicz (Eds.), *Handbook of Evolutionary Computation*, Chapter G3.7. Oxford University Press.

Huynen, M. and P. Hogeweg (1994). Pattern generation in molecular evolution: Exploitation of the variation in rna landscapes. *Journal of Molecular Evolution 39*(7), 1–79.

Ijspeert, A., J. Hallam, and D. Wilshaw (1997). Artificial lampreys: Comparing naturally and artificially evolved swimming controllers. In P. Husbands and I. Harvey (Eds.), *Proceedings of the Fourth European Conference on Artificial Life*, Cambridge, Mass, pp. 256–265. MIT Press.

Jakobi, N. (1994). Evolving sensorimotor control architectures in simulation for a real robot. Master's thesis, School of Cognitive and Computing Sciences, University of Sussex.

Jakobi, N. (1996a). Encoding scheme issues for open-ended artificial evolution. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel (Eds.), *Proceedings of the Fourth International Conferrence on Parallel Problem Solving in Nature*, Berlin, pp. 52–61. Springer-Verlag.

Jakobi, N. (1996b). Facing the facts: Necessary requirements for the artificial evolution of *complex* behaviour. Cognitive Science Research Paper CSRP422, University of Sussex.

Jakobi, N. (1996c). Harnessing morphogenesis. In *Proceedings of the International Conference on Information Processing in Cell and Tissue.*

Jakobi, N. (1997). Half-baked, ad-hoc and noisy: Minimal simulations for evolutionary robotics. In P. Husbands and I. Harvey (Eds.), *Proc. 4th European Conference on Artificial Life*, pp. 348–357. M.I.T. Press.

Jakobi, N. (1998). Evolutionary robotics and the radical envelope of noise hypothesis. *Journal of Adaptive Behaviour 6*(2), 325–368.

Jakobi, N., P. Husbands, and I. Harvey (1995). Noise and the reality gap: The use of simulation in evolutionary robotics. In F. Moran, A. Moreno, J. Merelo, and P. Chacon (Eds.), *Proc. 3rd European Conference on Artificial Life*, pp. 704–720. Springer-Verlag, Lecture Notes in Artificial Intelligence 929.

K-Team (1993, June). Khepera users manual. EPFL,Lausanne.

Kitano, H. (1995). Cell differentiation and neurogenesis in evolutionary large scale chaos. In *Proceedings of the European Conference on Artioficial Life*. Springer-Verlag.

Kodjabachian, J. and J.-A. Meyer (1994). Development learning and evolution in animats. In Gaussier and Nicoud (Eds.), *Proceedings from Perception to Action Conference*. IEEE Computer Society Press.

Koza, J. (1990). Evolution of subsumption using genetic programming. In F. Varela and P. Bourgine (Eds.), *Proceedings of the First European Conference on Artificial Life*, Cambridge, Massachusetts, pp. 110–119. MIT Press.

Koza, J. (1992). *Genetic Programming: on the programming of computers by means of natural selection*. Cambridge, Massachusetts: MIT Press.

Koza, J. and J. Rice (1992). Automatic programming of robots using genetic programming. In *Proceedings of AAAI-92*, Cambridge, Massachusetts, pp. 194–201. MIT Press.

Lund, H. (1995). Evolving robot control systems. In J. Alander (Ed.), *Proceedings of 1NWGA*, University of Vaasa,.

Lund, H. and J. Hallam (1996). Sufficient neurocontrollers can be surprisingly simple. Research Paper 824, Department of Artificial Intelligence, University of Edinburgh.

Lund, H., J. Hallam, and W.-P. Lee (1997). Evolving robot morphology. In *Proceedings of the IEEE 4th International Conference on Evolutionary Computation*. IEEE Press.

Mataric, M. and D. Cliff (1996). Challenges in evolving controllers for physical robots. *Robot and Autonomous Systems 19*(1), 67–83.

Michel, O. (1995). An artificial life approach for the synthesis of autonomous agents. In J. Alliot, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers (Eds.), *Proceedings of the European Conference on Artificial Evolution*. Springer-Verlag.

Miglino, O., H. Lund, and S. Nolfi (1995). Evolving mobile robots in simulated and real environments. *Artifical Life 2*(4), 417–434.

Miller, G. F. and D. Cliff (1994). Protean behavior in dynamic games: Arguments for the coevolution of pursuit-evasion tactics. In *Proceedings of the third international conference on Simulation of Adaptive Behavior*, pp. 411–421. MIT Press/Bradford books.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. Cambridge MA: MIT Press.

Nilsson, N. J. (1984, April). Shakey the robot. Technical Note 323, SRI International, Menlo Park, California.

Nolfi, S., D. Floreano, O. Miglino, and F. Mondada (1994). How to evolve autonomous robots: Different approaches in evolutionary robotics. In R. Brooks and P. Maes (Eds.), *Artificial Life IV*, pp. 190–197. MIT Press/Bradford Books.

Nolfi, S., O. Miglino, and D. Parisi (1994). Phenotypic plasticity in evolving neural networks. In P. Gaussier and J.-D. Nicoud (Eds.), *Proceedings of the From Perception to Action Conference*. IEEE Computer Society Press.

Nolfi, S. and D. Parisi (1995a). Evolving artificial neural networks that develop in time. In F. Moran, A. Moreno, and J. Merelo (Eds.), *Advances in Artificial Life: Proceedings of the Third European Conference on Artificial Life.*, Berlin, pp. 353–367. Springer-Verlag.

Nolfi, S. and D. Parisi (1995b). Evolving non-trivial behaviours on real robots: an autonomous robot that picks up objects. In M. Gori and E. Soda (Eds.), *Proceedings of Fourth International Congress of the Italian Association of Artificial Intelligence*, Berlin. Springer Verlag.

Nowak, M. and P. Schuster (1989). Error thresholds of replication in finite populations, mutation frequencies and the onset of muller's ratchet. *Journal of Theoretical Biology 137*(4), 375–395.

Pfeifer, R. and C. Scheier (1998). Embodied cognitive science: A novel approach to the study of intelligence in natural and artificial systems. In T. Gomi (Ed.), *Proceedings of ER'98*, pp. 1–36. AAI Books.

Press, W., W. Vetterling, S. Teukolsky, and B. Flannery (1992). *Numerical recipes in C (2/e)*. CUP.

Rumelhart, D. E., G. Hinton, and R. Williams (1986). Learning internal representations by error propogation. In D. Rumelhart and J. McClelland (Eds.), *Parallel Distributed Processing*, Volume 1, pp. 318–362. Cambridge, Mass: MIT Press.

Salomon, R. (1996). The influence of different coding schemes on the computational complexity of genetic algorithms in function optimization. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel (Eds.), *Proceedings of the Fourth International Conference on Paprallel Problem Solving in Nature*, Berlin, pp. 227–235. Springer-Verlag.

Schults, A. and J. Grefenstette (1992). Using a genetic algorithm to learn behaviours for autonomous vehicles. In *Proceedings of the American Institute of Aeronautics and Astronautics Guidance, Navigation and Control Conference*, Hilton Head SC, pp. 739–749. AIAA.

Seth, A. (1998). Scaling up evolutionary robotics: Noise and the pursuit of complexity. In P. Husbands and J.-A. Meyer (Eds.), *Proceedings of Evorob98*. Springer Verlag.

Sims, K. (1994a). Evolving 3d morphology and behavior by competition. In R. Brooks and P. Maes (Eds.), *Proceedings of Artificial Life '94.* M.I.T. Press.

Sims, K. (1994b). Evolving virtual creatures. In *Proceedings of Siggraph '94.*, pp. 15–22.

Smith, J. and T. Fogarty (1997). Operator and parameter adaption in genetic algorithms. *Soft Computing 1*(2), 81–87.

Smith, T. (1998). Blurred vision: Simulation-reality transfer of a visually guided robot. In P. Husbands and J.-A. Meyer (Eds.), *Proceedings of Evorob98*. Springer Verlag.

Smithers, T. (1994). What the dynamics of adaptive behaviour and cognition might look like in agent-environment interaction systems. In T. Smithers and A. Moreno (Eds.), *3rd International Workshop on Artificial Life and Artificial Intelligence, The Role of Dynamics and Representation in Adaptive Behaviour and Cognition*, San Sebastian, Spain.

Smithers, T. (1997). Autonomy in robots and other agents. *Brain and Cognition 34*(1), 88–107.

Thompson, A. (1995a). Evolving electronic robot controllers that exploit hardware resources. In F. Moran, A. Moreno, J. Merelo, and P. Chacon (Eds.), *Advances in Artificial Life: Proc. 3rd European Conference on Artificial Life*, pp. 640–656. Springer-Verlag, Lecture Notes in Artificial Intelligence 929.

Thompson, A. (1995b). Evolving fault tolerant systems. In *Proceedings of the first IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, Number 414, pp. 520–524. IEE conference publication.

Thompson, A. (1996a). Evolutionary techniques for fault tolerance. In *Proceedings of the UKACC International Conference on Control*, pp. 693–698. IEE.

Thompson, A. (1996b). An evolved circuit, intrinsic in silicon, entwined with physics. In T. Higuchi and M. Iwata (Eds.), *Proc. 1st Int. Conf. on Evolvable Systems (ICES'96)*, LNCS. Springer-Verlag. In press.

Todd, P. (1996). Sexual selection and the evolution of learning. In R. Belew and M. Mitchell (Eds.), *Adaptive Individuals in Evolving Populations: Models and Algorithms*, pp. 365–393. Reading MA: Addison-Wesley.

Todd, P. and G. Miller (1993). Parental guidance suggested: how parental imprinting evolves through sexual selection as an adaptive learning mechanism. *Adaptive Behaviour 2*(1), 5–47.

Vaario, J. (1993). *An Emergent Modelling Method for Artificial Neural Networks*. Ph. D. thesis, The University of Tokyo.

Viola, P. (1988). Mobile robot evolution. Bachelors thesis, M.I.T.

Wehner, R. (1987). Matched-filters - neural models of the external world. *Journal of Comparative Physiology 161*(4), 551–531.

Winston, P. (1988). *LISP* (3rd ed.). Adison Wesley.

Wolpert, D. (1995). No free lunch theorems for search. Working Papers List 95-02-010, Santa Fe Institute, CA.

Yamanuchi, B. and R. Beer (1994). Integrating reactive, sequential, and learning behaviour using dynamical neural networks. In D. Cliff, P. Husbands, J.-A. Meyer, and S. Wilson (Eds.), *Proceedings of the Third International Conference on the Simulation of Adaptive Behaviour*, Cambridge, Massachusetts, pp. 382–392. MIT Press/Bradfork Books.