

Digital control networks for virtual creatures

Christopher James Bainbridge

Doctor of Philosophy
School of Informatics
University of Edinburgh
2010

Abstract

Robot control systems evolved with genetic algorithms traditionally take the form of floating-point neural network models. This thesis proposes that digital control systems, such as quantised neural networks and logical networks, may also be used for the task of robot control. The inspiration for this is the observation that the dynamics of discrete networks may contain cyclic attractors which generate rhythmic behaviour, and that rhythmic behaviour underlies the central pattern generators which drive low-level motor activity in the biological world.

To investigate this a series of experiments were carried out in a simulated physically realistic 3D world. The performance of evolved controllers was evaluated on two well known control tasks — pole balancing, and locomotion of evolved morphologies. The performance of evolved digital controllers was compared to evolved floating-point neural networks. The results show that the digital implementations are competitive with floating-point designs on both of the benchmark problems. In addition, the first reported evolution from scratch of a biped walker is presented, demonstrating that when all parameters are left open to evolutionary optimisation complex behaviour can result from simple components.

Acknowledgements

“I know why you’re here... I know what you’ve been doing... why you hardly sleep, why you live alone, and why night after night, you sit by your computer.”

I would like to thank my parents and grandmother for all of their support over the years, and for giving me the freedom to pursue my interests from an early age.

My time at Edinburgh would not have been so enjoyable without my friends and flatmates; in particular I must thank J.D. for his friendship and kindred humour, Big C. for the nights of Teviot and snakebite, Ljiljana for her remarkable ability to share a room with me for three years and not go insane, Javier, Sasha and Jude for the similar skills of flat sharing, Nicki, Evie and Marian for being there, and everyone in the university ju-jitsu club for providing years of social interactions, the complexity of which, when graphed, might put a neural network to shame.

From my life before Edinburgh: Frank, John, Ian and Brian for the crazy nights in Consett, including the impromptu ju-jitsu demonstration that almost got us arrested.

And from a time long past: the inhabitants of the DnC BBS who gave me the inspiration and motivation to sit every night playing with code, networks, and Unix.

And to my younger brother, Paul, who as he points out, has had a job for a long time now...

Without various pieces of software this project would not have been possible. I would like to thank the authors of the following open source systems which I use every day: Linux, the Open Dynamics Engine (ODE) and PyODE, Screen, KDE and KPDF/Okular, R, Python, teTeX, Vim, ZODB/ZEO, Git, Firefox, and MPlayer.

This thesis was greatly improved thanks to valuable feedback from Gillian Hayes and Susan Stepney.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Christopher James Bainbridge)

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contributions to knowledge	7
1.3	Reproducibility	11
1.4	Summary	12
1.5	Organisation of the thesis	14
2	Biological sensing and control	17
2.1	Neural networks	17
2.2	Biological models	22
2.3	Connectivity and resilience	22
2.4	Pattern generation for muscle control	28
2.5	Sensory feedback	31
2.6	Brain computer interfaces	32
2.7	Summary	35
3	Synthetic neural networks	39
3.1	Similarities to biological networks	42
3.2	Connectivity	43
3.3	State and signal coding	44
3.3.1	Example codings	46
3.4	Models of single neuron dynamics	49
3.4.1	Input function	49
3.4.2	Activation function	50
3.4.3	Output function	50
3.4.4	Spiking models	52
3.4.5	Continuous models	57

3.4.6	Reduced models	61
3.5	Computational power	75
3.6	Robot control	76
3.7	Training	76
3.7.1	Backpropagation	77
3.8	Hardware acceleration	79
3.9	Summary	82
4	Other networks	83
4.1	Analog circuits	84
4.2	Digital circuits	85
4.3	Asynchronous circuits	86
4.4	Genetic regulatory networks	89
4.5	Boolean networks	94
4.5.1	Random boolean networks	96
4.6	Generalised logical networks	102
4.7	Cellular automata	106
4.8	The edge of chaos	110
4.9	Summary	111
5	Genetic evolution	115
5.1	Natural selection	116
5.2	Genetics in nature	117
5.3	Genetic algorithms	124
5.4	Modularity of the genome	126
5.5	Fitness function	127
5.6	Fitness surface	128
5.7	Search behaviour	131
5.8	Genotype encoding	132
5.8.1	Example encodings	134
5.9	Population models	135
5.10	Initial population creation	136
5.11	Parent selection	136
5.12	Reproduction	138
5.13	Summary	140

6 Evolution of specific genotypes	143
6.1 Problem specific operators	144
6.2 Evolving flocking	146
6.3 Ant colonies	148
6.4 Evolving neural networks for robot control	150
6.4.1 Timeline	152
6.5 Evolving reduced continuous neural models	157
6.6 Evolving reduced spiking neural models	158
6.7 Evolving modular hierarchical neural networks	158
6.8 Evolving genetic regulatory networks	161
6.9 Evolving cellular automata	161
6.9.1 Evolving cellular automata neural networks	164
6.10 Evolving analog circuits	168
6.10.1 Evolving FPAA robot control	168
6.10.2 Evolving FPGA oscillators	169
6.10.3 Evolving FPGA frequency discriminators	169
6.10.4 Evolving on an analog FPTA	170
6.10.5 Evolving fault tolerance	171
6.11 Evolving digital circuits	172
6.11.1 Optimising gate count	173
6.11.2 Optimising power	174
6.11.3 Evolving digital circuits for robot control	176
6.12 Evolving pattern generators	176
6.12.1 Timeline	177
6.13 Evolving morphology	180
6.13.1 Timeline	181
6.14 Evolving robot morphology and control	193
6.14.1 Timeline	194
6.15 Evolving modular robots	211
6.16 Summary	212
7 Overview so far	217
8 Software design	219
8.1 Creature morphology	219
8.2 Morphogenesis	221

8.3	Evolution of the morphology	223
8.4	Example morphologies	224
8.5	Neurogenesis	225
8.6	Neural network topologies	233
8.7	Neuron quantisation	236
8.8	Evolution of the neural networks	237
8.9	The software	238
8.10	Simulated physics	244
8.10.1	Collision detection	245
8.11	Sensors and actuators	246
8.12	Motor models	246
8.12.1	Stimulus-response curves of PD motor controller	247
8.13	Physics simulation problems	250
8.14	Tasks	256
8.15	Summary	256
9	Software testing	257
9.1	Testing neuron models	259
9.1.1	Explanation of graphs	260
9.1.2	Sine model	262
9.1.3	Sigmoid model	264
9.1.4	Beer model	267
9.1.5	Taga model	270
9.1.6	Ekeberg model	273
9.1.7	Integrate-and-fire model	283
9.1.8	Spike response model	286
9.1.9	Logical model	290
9.2	Cluster performance	292
9.3	Summary	295
10	Pole balancing experiments	299
10.1	Introduction	299
10.2	Task	302
10.3	Task analysis	302
10.4	LQR controller design	303
10.5	Experimental design	304

10.6	Reproducibility	308
10.7	Results	308
10.7.1	ANOVA modelling	308
10.7.2	ANOVA results	312
10.7.3	About “Least Significance Difference” plots	313
10.7.4	“Least Significant Difference” plots	314
10.7.5	Factor: Model	315
10.7.6	Factor: Quanta	316
10.7.7	Factor: Number of generations and population size	317
10.7.8	Factor: Timing	318
10.7.9	Factor: Interaction of model and timing	319
10.7.10	Factor: Interaction of neuron model and generations / population size	320
10.7.11	Factor: Interaction of neuron model and quantisation	321
10.8	Summary	331
11	Virtual creature experiments	333
11.1	Introduction	333
11.2	Task	335
11.3	Fitness function	335
11.4	Experimental design	338
11.5	Reproducibility	339
11.6	Results	339
11.6.1	ANOVA modelling	339
11.6.2	ANOVA results	342
11.6.3	“Least Significant Difference” plots	342
11.6.4	Factor: Model	343
11.6.5	Factor: Number of generations and population size	344
11.6.6	Factor: Timing	345
11.6.7	Factor: Interaction of neuron model and timing	346
11.6.8	Factor: Interaction of neuron model and quantisation	347
11.7	Example evolved control	355
11.7.1	An explanation of these graphs	355
11.7.2	Example floating-point controller (Beer’s CTRNN model) . .	357

11.7.3	Example quantised controller (Beer's CTRNN model with 16 quanta states)	360
11.7.4	Example quantised controller (sigmoid model with 8 quanta states)	362
11.7.5	Example quantised controller (Ekeberg model with 4 quanta states)	364
11.7.6	Example quantised controller (integrate-and-fire model with 4 quanta states)	366
11.7.7	Example quantised controller (SRM model with 16 quanta states)	368
11.7.8	Example quantised controller (Logical model with 2 quanta states)	373
11.7.9	Example quantised controller (Sine model with 16 quanta states)	375
11.8	Example of evolution — from biped walking onwards	377
11.8.1	A note on lack of reproducibility	378
11.8.2	The observed evolution	379
11.8.3	Stage 0 — walking biped	380
11.8.4	Stage 1 — smaller head, pushing forward on knees	381
11.8.5	Stage 2 — loses a leg	382
11.8.6	Stage 3 — stronger leg	383
11.8.7	Stage 4 — better global coordination, longer leg	384
11.9	Summary	385
12	Conclusions	389
12.1	Summary	389
12.2	Future work	391
Bibliography		395

Chapter 1

Introduction

1.1 Motivation

Autonomous robots are increasingly being used for applications which humans find to be dull, dangerous or economically inefficient. The prime motivation in modern robotics research is to develop new techniques for building robots that are cheaper, faster, and more intelligent than today's robots. Such robots are used in fields as diverse as manufacturing, space exploration, and entertainment (e.g. figure 1.1).

Since the early days of robotics, research in the field has been split into two separate camps; that of hardware design, which has focused on the materials, sensors and actuators of an embodied robot, and that of controller design, which has focused on the attainment of goals through the planning and carrying out of actions. In many ways this division has mirrored the academic differentiation between psychologists, who study the mind, and biologists, who are more concerned with the practical functionality of the body.

Manufacturing robots has traditionally been an expensive and time consuming process (figure 1.2). A human designer must specify in intricate detail the form and function of each robot part, and then integrate them into a functioning whole. Each part must be built, and the robot assembled. Software must then be designed and coded. Only at the end of this design cycle can the robot be fully tested to see whether it fulfils its design objectives. Errors can be introduced at any stage, and usually will not become evident until later in the testing stage. For each error the cycle must be iterated, thus multiplying the development time.

The coming age of nano-manufacturing promises low cost production of even the most complex mechanical parts [114, 162]. It is already possible to print arbitrary

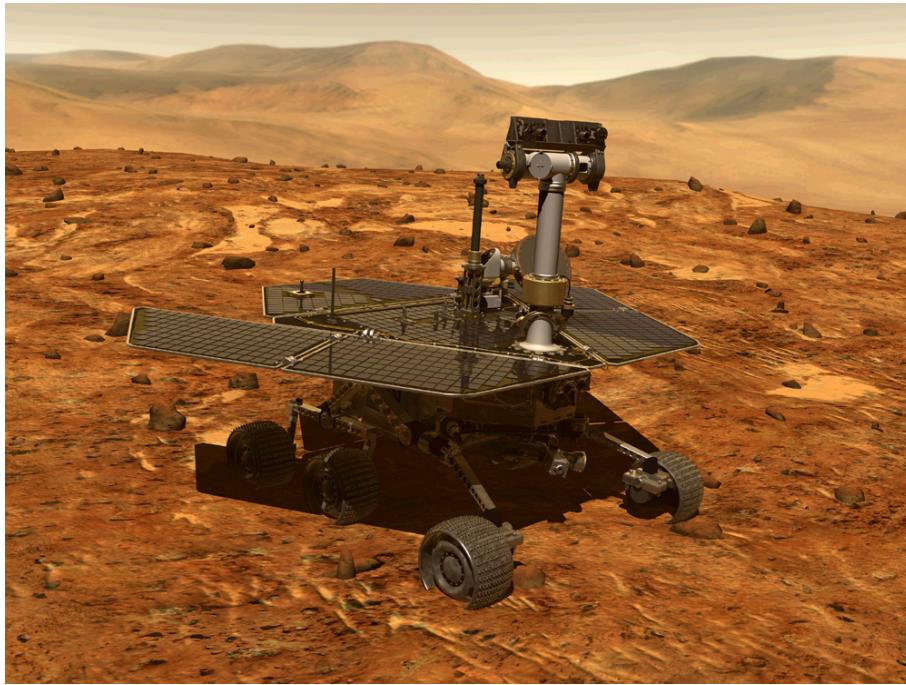


Figure 1.1: *Semi-autonomous robotic systems are increasingly being used in space. NASA's "Spirit" rover has been exploring the surface of Mars since January 2004.*
Credit for image: NASA [221]

3D structures containing complex parts such as batteries and actuators [274]. If this comes to pass we will see robots with complex bodies, displaying physical abilities that match those of humans. We already have biped robots that are capable of walking, climbing, dancing and performing somersaults (figure 1.3). An advancement as great as that of nano-manufacturing would enable the construction of synthetic bodies that are stronger, faster and more reliable than those from the world of nature. However, the true challenge would then be to understand and create synthetic intelligence that can rival that of biological creatures, displaying the same complex behavioural, reasoning and communication skills.

Biologically inspired robotics draws inspiration from the natural world. By analysing the bodies and nervous systems of living creatures we hope to gain understanding and knowledge that can then be applied to the production of synthetic creatures. We often find that biological creatures have evolved very efficient designs that operate in a manner alien to that of a human designer. When human designers are faced with an explosion of complexity they employ abstractions to simplify the design space. Nature, knowing no such bounds, tends to create designs with many interconnected recurrent hierarchies, making their operation complex and difficult to analyse.

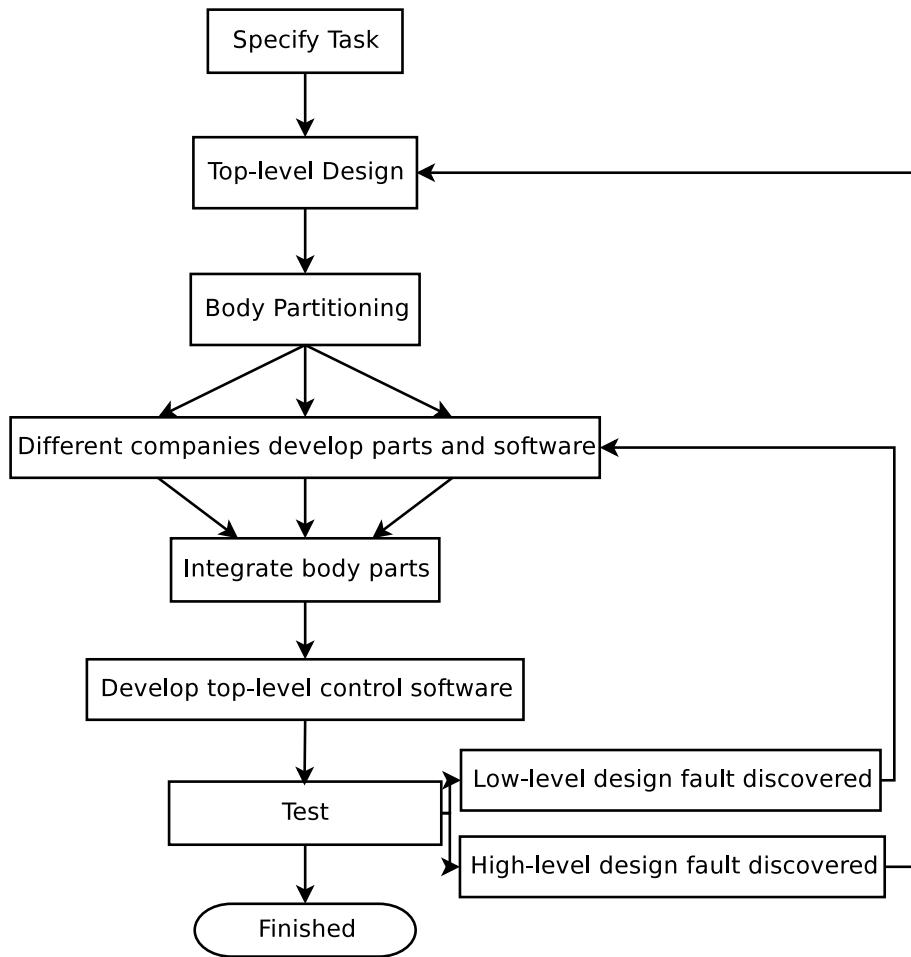


Figure 1.2: *A typical design methodology for commercial robotics.*

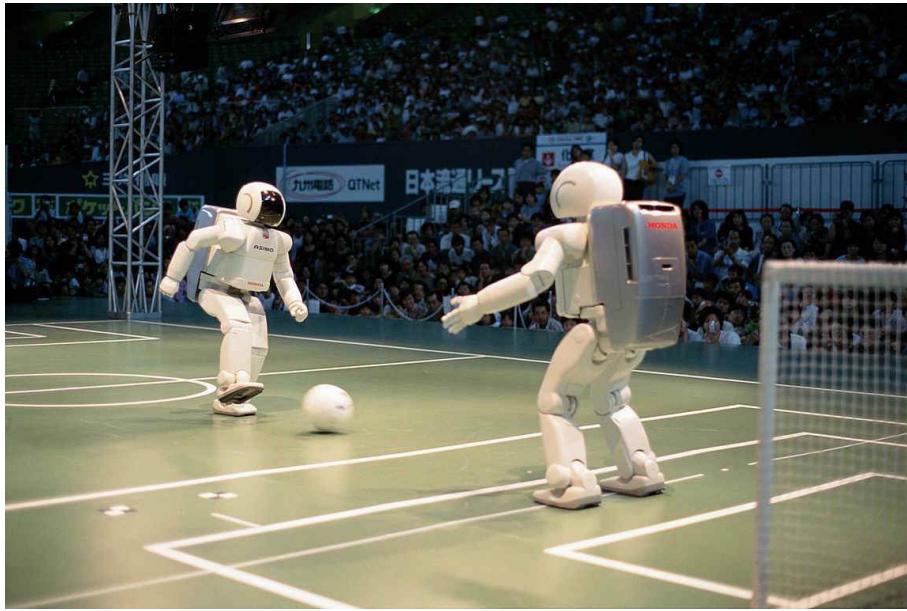


Figure 1.3: Honda's ASIMO represents the pinnacle of current robotic engineering. It can run, dance, climb stairs, recognise faces, respond to over 200 spoken words, and locate objects pointed to by a human. Its motions, however, appear unnatural; its “zero-moment point” controller can not move through dynamically unstable positions.

Credit for image: HONDA [190]

What we seek are new ways to design successful robots. The difficulties faced by software designers in recent years suggest that we have reached the limit of what human programmers are capable of — in the words of Winograd:

The symbolic paradigm... has turned out to be a dead end... In order to build human-like intelligence, researchers will need to base it on a deep understanding of how real nervous systems are structured and how they operate. ([425])

We find it difficult to analyse complex, non-linear systems, and find it impossible to reason about systems that contain millions of interconnected, dependent variables — as Dijkstra said:

When all is said and done, the only thing computers can do for us is to manipulate symbols and produce results of such manipulations. From our previous observations we should recall that this is a discrete world and, moreover, that both the number of symbols involved and the amount of manipulation performed are many orders of magnitude larger than we can envisage: they totally baffle our imagination and we must therefore not try to imagine them. ([109])

The design of complex robot controllers in the future is likely to require the use of automated design techniques that are able to thrive where humans have failed.

One of the approaches that has been successfully employed to automate robot controller design is that of artificial evolution. The diversity and success of the living world spring from a process of evolution and survival of the fittest, carried out over millions of years. By emulating the critical aspects of this process we can create an environment where “good” solutions can thrive and reproduce, whilst “bad” ones die off. Evolutionary development occurs as a pseudo-random search through the design space, driven by a genetic algorithm and a fitness function that separates the good from the bad.

Unlike hand-crafted controllers, evolved controllers tend to perform well on dynamic, real-world physical tasks. The combination of a noisy world, random genotype mutations and survival of the fittest produce controllers that are more robust to sensor error and component failure. Human designers tend to create highly centralised designs. In contrast, evolved controllers are highly decentralised, and therefore less prone to failures in individual components. Evolved controllers also tend to display more properties attributable to self-organisation; with evolved neural networks in particular showing a robustness to initial starting conditions that is not present in hand-crafted software.

In robotics power usage is a great concern. Modern high-performance processors are notoriously power hungry; the Intel dual-core Xeon processor has a “thermal design power” (the amount necessary for sustained maximum operation) of 165 Watts [105]. In contrast, the bodies of creatures are incredibly power efficient. The human body has an average consumption of around 100 Watts, with around 10-20 Watts attributable to the brain [115, 222]. The increased energy efficiency of biological brains is due to their distributed nature, which enables the carrying out of highly parallel processing at a low switching frequency in the tens of Hertz.

There is a great amount of interest in attempting to predict when the computational power and storage capacity of AI systems will approach that of the human brain. Digital computers and biological brains operate in a completely different way, and hence it is difficult to compare them. Despite this, it has been estimated that the computational power of the human brain is between 10^{13} and 10^{16} operations per second [291]. This agrees with other estimates which place the figure around 10^{11} [306] to 10^{14} [311].

The total storage capacity is estimated to be 119 megabytes of specific pieces of information about the world [290]. This figure is based on experimental studies with

human subjects being asked to remember and recall information, and ignores how the data is actually coded and represented at lower levels. Storage capacity, at the level of individual neurons, or even proteins, is estimated to be around 100 million megabytes [311]. If these estimates are correct, and computational power continues to increase according to Moore’s Law, then we should expect it to equal that of the human brain sometime after 2020 [53, 474]. Of course, this is merely an estimate of raw power, and if we fail to comprehend how the brain works, then we will be unable to translate this into a measure of intelligence.

Neural controllers consist of collections of very simple processing elements, which as a whole display properties that are greater than the mere sum of their parts. This synergistic property is commonly referred to as emergence, and has been a subject of interest amongst the artificial intelligence research community for a long time. It is hoped that evolved controllers will similarly display complex behaviours that are somehow greater than the mere switching of states orchestrated by individual neurons.

It is highly desirable for robots to be autonomous — that is, to be able to operate independently of a human operator. The state of the art is currently to be “partially autonomous”, where a human planner instructs the robot to perform some series of simple tasks, and the robot controller performs a small amount of decision making in order to satisfy each task. This kind of system is essential in areas such as space and planetary exploration, where the latency of the communication medium between the robot and operator is too great for real-time control.

The ability to learn from past actions and experiences would both enable better functional performance, and save the human designer from the problems of having to anticipate all situations which the robot may face over its lifetime. The use of genetic algorithms can be viewed as a form of static learning, with knowledge of the fitness evaluation task being embedded within the genomes of the evolved creatures. There are various algorithms for the online updating of neural networks when faced with classification style tasks, however, it has proven much more difficult to perform this kind of online learning with planning and control systems. At a low level, robots like Sony’s AIBO can utilise genetic algorithms to adapt their signals to compensate for manufacturing variances, and component degradation through wear and tear, and even the motor levels necessary to drive walking gaits [199], but more complex control tasks such as facial recognition and task planning still rely on pre-programmed non-adaptive algorithms.

1.2 Contributions to knowledge

This thesis explores the use of quantised neural networks for the task of robot control. Quantisation is the process of converting a real value into a discrete value drawn from a finite set, by mapping regions of the domain onto single values. In signal processing, quantisation is used to convert a continuous value, often gathered by an analog sensor, into an integer with a specified precision. An example of quantisation would be the use of the floor function to map a continuous value to an unsigned 8-bit integer: $y = \lfloor 255x \rfloor$ where $x \in [0, 1]$, producing the 8-bit output $y \in \{0, \dots, 255\}$. For a signal consisting of a sequence of continuous values, the quantisation function will map the signal onto a sequence of discrete symbols, often an integer series. A computational or electrical system that uses discrete values is known as a “digital” system. In electronics, the quantisation function is usually carried out by an analog-to-digital converter.

A related aspect to value continuity is the concept of temporal continuity. In the real world, sensors can represent a sensed state using a continuous-time signal — that is, a signal that varies in response to all perceivable permutations in an input signal. This continuous-time signal can be mapped onto a sequence of discrete values by sampling at some given frequency (the exact frequency is known as the “sampling rate”). The vast majority of modern digital processors are synchronous, meaning that the action of changing state is coordinated between the internal state-holding elements so that it happens at a single point in time, making state change a global event rather than local. The number of times this event occurs per second is defined by the frequency of the synchronous clock. The twin properties of a modern processor being synchronous and digital imply that a continuous-value continuous-time input signal from a sensor must be both quantised and sampled before being presented to the processor.

It is well known that continuous connectionist architectures, such as those used to emulate neural networks, can be used for dynamical robot control [147, 293, 349]. The use of networks of interconnected digital nodes, each with a discrete number of states, has been less-thoroughly researched. There is no successful methodology for designing large, complex, digital network robot controllers, and we do not know how the performance of such controllers would relate to that of continuous neural controllers. Other digital connectionist architectures, such as random boolean networks, asynchronous circuits, and cellular automata, will also be discussed.

Neural network research has traditionally relied on the simulation of continuous dynamical systems using digital processors [491]. The von Neumann architecture used

in modern computers does not lend itself to high speed simulation of massively parallel networks, as all processing must be reduced to a series of sequential operations. With synchronous updating, the processor must loop over every node in the network, calculate its next state, and store this state somewhere. Only after all of the next states have been calculated can the network be updated. This scheme gives the appearance of nodes being updated simultaneously, and prevents the ordering of individual node updates from affecting the overall system behaviour, but it also enforces serialisation of neuron processing. Operations are slowed by a high memory latency between the CPU and main memory; there are few cache hits due to the need to process the rest of the network between individual neuron updates, and the memory required for a large network exceeding the size of the data cache.

These problems suggest that von Neumann processors are not suited for the simulation of large neural networks. There have been attempts to utilise multiple CPUs [218, 314, 323], custom neural network processors [128, 287], DSP arrays [315] and FPGA arrays [185], but they all share a common problem in that they attempt to preserve the way that simulation is currently done, with floating-point (or analog) values. Calculating neuron state updates requires either a floating-point unit or dedicated analog circuits. Both are expensive in terms of area and energy.

High throughput floating-point units consume a large amount of power, and take up a considerable amount of space on the CPU die. Analog circuits simulating individual neurons tend to be smaller, since there are usually a small number of neurons, but still require a far greater number of transistors than individual digital logic gates. This means that there are fundamental problems in scaling this style of architecture up to the requirements of simulating the 100 billion (10^{11}) neurons and 100 trillion (10^{14}) synapses of the human brain.

The simulation of spiking neural networks is unique in that the values transferred between neurons are digital, with information being coded in either the frequency or timing of pulses. However, the neuron update function still relies on floating-point or analog operations.

One potential solution to the scaling problem is to examine whether continuous simulation is necessary in the first place. The observation that current research is carried out using digital computers implies that it might not be. Modern floating-point units comply with the requirements of the ANSI/IEEE 754 standard for binary floating-point arithmetic [159], which specifies that floating-point numbers are encoded as either single (32-bit), or double (64-bit) data types. There is some redundancy in the

encoding, so the true precision of these floating-point data types is actually 24-bit for single and 53-bit for double. Despite the fact that we use these data types to store real numbers, they are in fact quantised values and are subject to rounding errors due to the inability to encode true continuity.

Since we know that these systems are capable of successfully simulating neural networks, we have reason to believe that reducing the precision by further quantisation, ultimately to a two-valued boolean system, may yield some success. Evidence from biology, where continuous genetic networks are routinely simulated using quantised abstractions [95], also suggests that continuity is not a pre-requisite for accurate simulation of dynamical network behaviour, although the opposite position — that binary systems are unable to reproduce the complex dynamics of continuous gene expression — has also been argued [371].

There have been attempts to reduce both the continuous and spiking models by removing the requirement for floating-point arithmetic in order to enable more efficient implementations with digital logic (see section 3.4.6.3). However, these models have seen little use. Some have only been proposed, and never tested on real world applications. Only a few have been directly compared to continuous models, using some static classification or filtering task, and nothing dynamic like robot control. Only one of the reduced spiking models has been used for 3D robot control. So far, there is no evidence that these models can carry out more complex robot control tasks, and no quantitative analysis of how they compare to their continuous counterparts.

If quantised neural networks can be built that successfully function as robotic controllers, as continuous networks can be now, then we will be able to drastically reduce the requirements of hardware implementations, and thus enable scaling up to the massively sized networks we see in the human brain.

The results of research in this area are potentially of great interest. Spivey has claimed that the continuity of the real world, both in terms of continuous state spaces within neural systems, and of temporal continuity, are necessary prerequisites for intelligent life [407, 408]. One possible argument against this is the phenomenon of temporal induction (or auditory continuity), in which subjects perceive a tone as being continuous despite the insertion of temporal gaps, which would imply that there is some mechanism within the brain that constructs the perception of continuity from discrete sensory input [209].

Penrose argues that the brain is a quantum computer, and hence we cannot hope to simulate it digitally [330]. Siegelmann argues that recurrent analog systems have

a greater computational power than digital systems [396]. Others have claimed the contrary; that neuronal spiking is actually a digital abstraction discovered by nature to provide a more reliable and accurate method of long distance signalling [96], and hence nature itself abstracts above the continuity of the physical world, suggesting the possibility that intelligence could be simulated with digital computation.

Some have gone even further in these musings; one hypothesis of note being Wolfram's conjecture that the physical fabric of the universe is in fact a discrete cellular automaton [480], that operates on the "edge of chaos" [254]. If true, this would certainly assure us that complex behaviours can arise within digital systems.

Recently, Hogan has proposed that the universe may in fact be a two-dimensional binary system [193]. The hypothesis is based on the concept that information can not be destroyed, and hence all of the physics that occurs in the universe will have an equivalent representation that occurs on the boundary of the event horizon of the cosmos. The event horizon of the cosmos is the two-dimensional manifold beyond which light has not yet had time to reach us since the beginning of the universe; i.e. the two-dimensional surface of an approximately spherical object with a radius of 13.7 billion lights years from the centre of the universe. In order for this to be true, the amount of information available at the surface of the cosmos horizon would have to be equal to the amount of information contained within the volume. Hogan has proposed that this could occur if the resolution of the smallest bit of effective information within the volume were lower than that on the surface. On the surface, the resolution of each unit would be 10^{-35} metres long (one Planck length), a unit which is too small to test for. Within the volume, the resolution of the smallest unit would be around 10^{-16} metres long, which would make it large enough to be detected by current equipment. Hogan has proposed that recent experimental results from the GEO600 gravitational wave detector match his predictions, and hence that the physics of the universe can be explained as two-dimensional computations, with the three-dimensional world we perceive being a blurry holographic projection of interactions occurring across the two-dimensional manifold [69].

Banerjee has claimed that biological neural systems are governed by chaotic attractors [13]. It would be an important step forward to show that digital systems such as cellular automata, which are known to be capable of displaying chaotic and semi-chaotic behaviours, are also capable of utilising this behaviour in a productive way to form a coherent computational system that can drive robot behaviour.

At the moment these claims are lacking in evidence either way. This research shows

that, for certain robot control tasks, discrete connectionist architectures can display the same complexity of behaviour as continuous systems, and quantifies the degradation in performance due to reduced precision. This not only has direct applications in the field of robotics, but may also provide insights into biological reasoning, signalling and control. This research will not answer any grand claims about the underlying nature of the fabric of the universe, however the above arguments do at least suggest the possibility that intelligent behaviour could be described by a digital computational system.

1.3 Reproducibility

The research underlying this thesis has relied on computer simulation, using custom written software combined with common libraries, to carry out experiments and gather data. The question arises as to how one can be sure that bugs in the developed software do not affect the validity of the research findings. The unfortunate answer is that there is no way that one can be certain of this; despite following software engineering best practices, it is still possible that there are bugs in the developed software. It is also possible that there are bugs in the libraries that this software relies on, such as the “Open Dynamics Engine” physics simulator.¹

A unit testing regime was followed in the development of the software (see page 240) and neural network models (see page 257). Unit tests exercised the critical code paths of the software, and the output was examined visually to see if there were any apparent problems. This mostly involved looking at 3D visualisations of physics simulations in order to see if anything was obviously incorrect, and in manually studying output signal data related to input change events to see if the model reacted in the expected way. The unit tests did catch several programming errors, both errors in the developed code and errors in the libraries that were being relied upon. These problems were fixed, and the final code used for the experiments passes all of the unit tests.

As noted in the relevant sections (p. 240 and p. 257), there is no way to *guarantee* that the software is bug free. No amount of testing will exercise every possible code path and data set that the various programs may execute or interpret. The problem is not unique to this research — all research that relies on software and simulation to gather and process experimental data faces the same issues.

¹One example of this fragility is that the floating-point neural network controllers evolved in chapter 11 display different behaviour on ODE if it is compiled to use 32-bit arithmetic in its simulations.

As software simulation has become more widely used in modern scientific experimentation, this has become an issue of increasing importance. Experiments may be carried out and findings published, only for software problems to be discovered that will later invalidate the research. In many cases, the custom software used to carry out experiments is not published, meaning that there is no way to exactly reproduce the research. When experiments rely on complex software, such as a structural model of a microprocessor with timing information back-annotated from layout, it is not only expensive and costly to design and build an alternative microprocessor, but due to unknown design differences, it is also highly unlikely that any reproduced experiments would generate exactly the same data anyway. When the original source code and experimental data is not made available, it is difficult for the traditional scientific process of independent external auditing and verification to take place.

This problem of reproducing research carried out through the development and use of complex software has been addressed by some notable scientists [51, 246, 253, 386, 458]. Vandewalle writes:

For a computational algorithm, details such as the exact dataset, initialization or termination procedures and precise parameter values are often omitted in the publication for various reasons. This makes it difficult, if not impossible, for someone else to obtain the same results. ([458])

These authors argue that only publication of the complete source code addresses such concerns — to quote Donaho:

An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures. ([51])

To this end, the source code written as part of the research for this thesis, and the data gathered, will be made available together with the thesis. As already pointed out, this does not guarantee that the software is bug free, but it does at least provide an assurance that future experimenters can analyse the source and engage in their own experiments with it.

1.4 Summary

Modern robot control systems often utilise neural network models which are continuous in both neuron state and time. The central hypothesis of this thesis is that such

detailed models may be unnecessarily complex, and that we may be able to use simpler models that are discrete in both state and time. The thesis will explore how these simpler models affect the ability of the controller to perform robot control tasks.

Two sets of experiments were devised to test the hypothesis. The experiments provide a “classic” AI robot control task which can be used to measure the performance of a controller by measuring some quantitative characteristic. The first task was the “pole balancing” problem, and the second task was locomotion control of an evolved robot. Both are common control tasks in AI research, with many papers having been published on both topics. Pole balancing is a more traditional research area, having been studied since the early days of AI. The study of robot locomotion has similarly been studied for a long time, however, controlling evolved creatures is a different challenge to that of the design of a controller for a fixed robot architecture, and has been studied for less time. The principal difference is that a human designer will typically have complete control over the specifications of both hardware and software for a traditional design, whereas an evolution based solution utilising genetic algorithms will search through a complex design space automatically.

The first set of experiments test the performance of a neural network with a single output connected to the angular motor of a simple pole balancing robot. This is the traditional AI “pole balancing” problem, and the aim is to balance the pole for as long as possible. The second set of experiments use a combined set of neural networks to control a virtual creature, with the virtual creature being evolved alongside the networks in a unified genome.

In both experiments, several neuron models from the literature were implemented and tested. For each model, two versions were created; one, a continuous model, relies on high-precision floating-point arithmetic, the other, a digital model, relies on quantised low-precision integer arithmetic. These models were compared over the control tasks in order to draw conclusions about the use of continuous models versus digital models.

The neural models implemented were “continuous time recurrent” neurons as described by Beer, lamprey eel neurons as described by Ekeberg, a network of nodes implementing digital logic functions, the “integrate-and-fire” and “spike response” spiking models, the sigmoid model, and a timing-synchronised sine wave model.

Since we have no design principles for creating complex neural networks for control tasks, genetic algorithms were instead used to “evolve” solutions. In the first set of experiments, the network was evolved whilst the morphology (pole balancer) was

fixed. In the second set of experiments, the morphology of the virtual creature was evolved alongside the controller, resulting in robots that differed in both body and control.

The results showed that quantised networks could perform as well as continuous networks on the tested robot control tasks. Synchronous networks tended to perform better than asynchronous ones. Simple binary oscillators with minimal synchronisation and no sensory input performed better than complex floating-point neural models on the given tasks. This surprising success suggests that it is synchronised recurrent pattern generation, and not input signal processing, that is important for these particular control tasks. This leaves open the question of the best way to incorporate input signals into such an architecture, suggesting that at lower levels of control, input signals are not useful in generating basic activation patterns. For tasks where sensing is clearly necessary, such as scent-directed motion, it may be better to build, or to focus evolution towards, hybrid systems of distinct low-level pattern generators and high-level controllers, in which the low-level pattern generators are perhaps isolated from sensory input.

The conclusion of this thesis is that simple binary or integer models will suffice and perform just as well, or even better, than complex floating-point neural models for some common tasks where the more complex models would normally be deployed. This has a direct application in mobile robotics, in which controller power consumption is a major concern. The ability to use simpler models directly translates to the ability to use simpler, low-power processors, which reduce design complexity by eliminating floating-point units and reducing arithmetic precision. The experimental robot control tasks did not require sensory input; an obvious extension to this research would be to incorporate exteroceptive sensors into the robot morphology and redo the experiments, analysing whether a hybrid network approach joining binary oscillators with computational elements would be successful.

1.5 Organisation of the thesis

Chapter 2 describes how sensing and control work in real biological creatures. Animals process large amounts of input sensory data through massive neural networks, and in turn output signals which control the contraction and relaxation of muscle fibers. A great deal of work by biologists and neuroscientists has gone into understanding the mechanisms behind these processes. In particular, it is believed that repetitive

patterns of muscle activation, such as those necessary for movement based behaviours (e.g. walking, swimming), are generated by neural networks known as “central pattern generators”.

Chapter 3 describes how researchers have recreated synthetic neural networks in order to both learn more about the biological ones, and also to utilise them for real world applications. Various neuron models have been proposed which vary in complexity. Various network models have been proposed which vary in topology and connectivity. These networks are often used for pattern recognition tasks, and so it is necessary to “train” them with an appropriate data set in order to recognise and differentiate between various input patterns. The actual implementation medium also differs — synthetic neural networks have been simulated using regular computer systems, distributed clusters of servers, digital signal processors and other custom semiconductor designs.

Chapter 4 covers non-neural network designs, such as boolean networks and cellular automata. These network models share many properties of traditional neural networks and are often capable of carrying out the same tasks, but differ in levels of node state, quantisation, connectivity, timing, and inter-node signalling.

Chapter 5 provides an overview of the theory of evolution, genetics, and adaptability. The study of “genetic algorithms”, in which theories of genetic evolution are utilised in order to solve real world problems, is described, along with some cases where it has been applied successfully.

Chapter 6 covers real world cases where genetic algorithms have been used to evolve solutions to problems. This includes the evolution of various kinds of neural networks, cellular automata, electronic circuits, and virtual creatures. In each of these cases, the defining characteristics of the genetic algorithm are the solution genotype, morphogenesis, and the mutation operators applied. Each of these are specific to the problem domain. The use of genetic algorithms on a wide range of problems shows that the basic theory is both useful and adaptable. The problems discussed are all related to the topics of this thesis - the evolution of control networks for robots, rhythm generation with evolved digital circuits, evolution of static morphologies, evolution of morphology and control for complete robots, and the evolution of transistor-based analog and digital circuits. This chapter ends the background literature review.

Chapter 7 provides an overview of the thesis so far, including the findings of the literature review, and the aims, hypotheses, and experiments to be carried out.

Chapter 8 provides an overview of the software written for this project. Rather

than just cover the implementation in terms of architecture and features, the description also provides details of many design parameters, such as the genotype, morphogenesis, fitness tasks, and problem domains that the software can be configured to evolve solutions for. Overviews of the underlying physics and distributed database models are also included.

Chapter 10 describes the first set of experiments run, which compared the performance of evolved floating-point and quantised neuron models on the pole balancing problem. This is followed by results and analysis.

Chapter 11 describes the second set of experiments run, which compared the performance of evolved floating-point and quantised neuron models in generating locomoting behaviour in simulated, physically accurate virtual creatures. This is followed by results and analysis.

Chapter 12 summarises the thesis, draws conclusions, and suggests potential avenues for future research.

Chapter 2

Biological sensing and control

In order to recreate intelligence, we must first study it in nature. Biological creatures consist of complete, animated bodies, with sensory interpretation and muscle control being generated by complex neural networks within the spinal cord and brain. This chapter will describe the structure of the brain and its computational processing, and how current theories suggest repetitive cyclic motions, such as walking and swimming, are created by collections of neurons known as “central pattern generators”.

2.1 Neural networks

A neural network is a collection of simple processing units known as neurons. They are found in the central nervous system and brains of living creatures, and collectively perform all of the sensing and control behaviour displayed by the creature. Figure 2.1 shows part of a biological neural network.

At the lowest level of control sophistication lie simple creatures which can only react to their environment through reflex actions. Sensor neurons receive stimulus from the environment, which is then processed by intermediate neurons before being turned into actions by motor neurons which are connected to muscle fibers. These simple static feed-forward networks provide enough processing power to account for the reflex arc present in animals.

Simple reflex and cyclic actions rely on the dynamics of the network alone to create repetitive attractor patterns. More sophisticated creatures have complex neural networks which utilise feedback and dynamic adaptation of neuron and connection properties. These networks are capable of storing information, and can adapt their processing to better control the creature’s body.

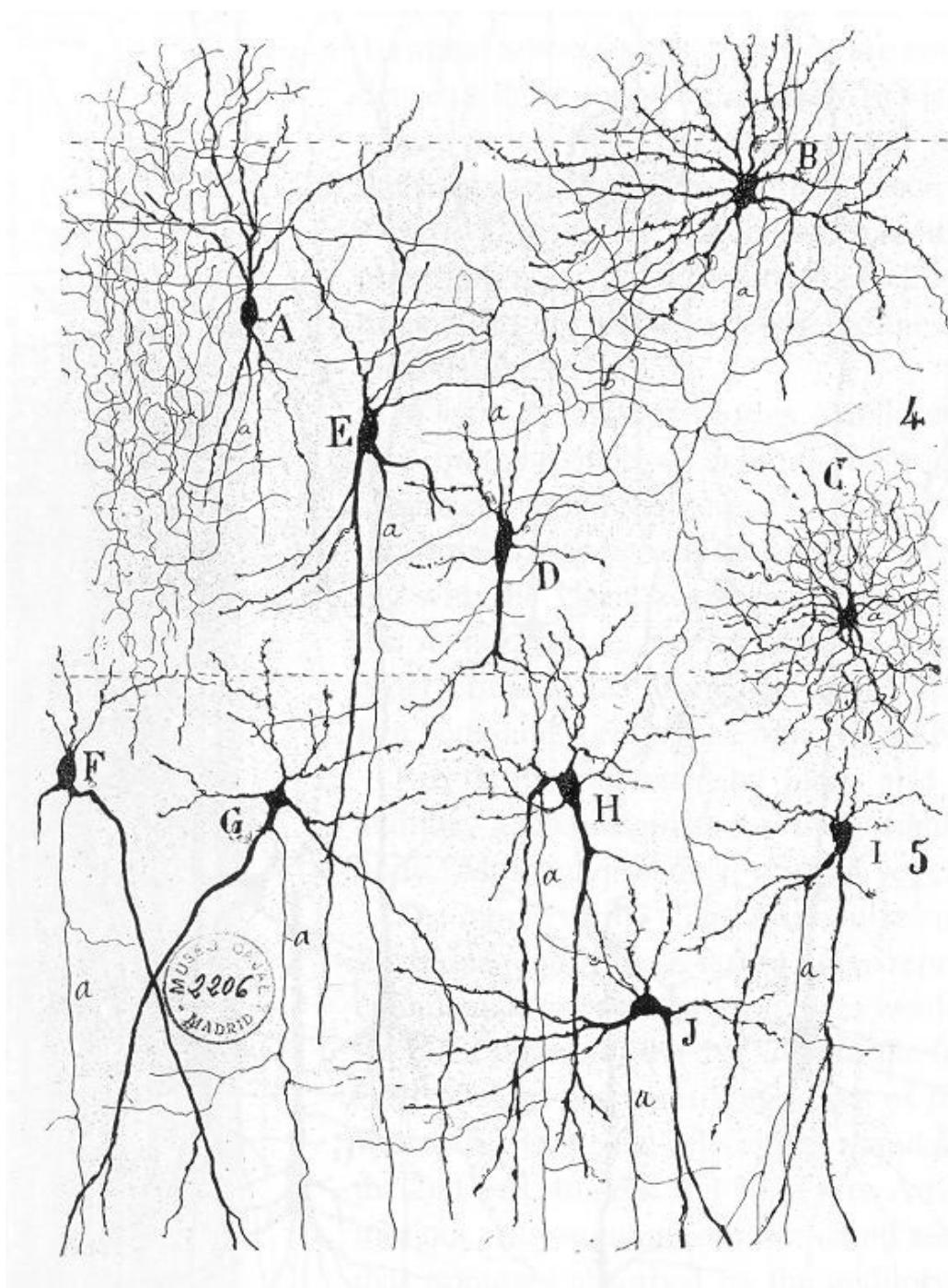


Figure 2.1: Part of a biological neural network, in this case, the auditory cortex. Each “blot” is an individual neural cell.

Credit for image: Santiago Ramón Y Cajal [54]

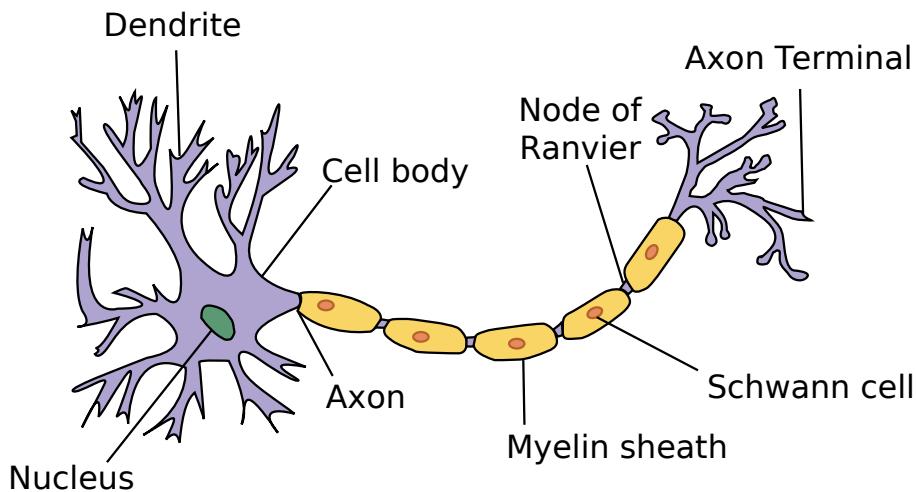


Figure 2.2: A biological neuron. Dendrites gather incoming connections from other neurons, some processing occurs inside the cell, and generated spikes travel outward along the axon. One neuron is connected to another via a synapse between an axon terminal and a dendrite.

Credit for image: Quasar Jarosz / U.S. National Cancer Institute

There are many different types of neuron which can be differentiated by the computation that they perform, and by the type and synthesis rate of neurotransmitters which their synapses release. Interestingly, stem cell research has shown that neural stem cells can morph into any type of neuron found in the nervous system [121, 126]. Understanding how these stem cells organise and structure the formation of the brain may lead to further insights into its functioning. Although it was thought for a long time that neuronal neurogenesis (the creation of new neurons) does not occur in adult mammals, since 1998 we have known that this does indeed occur, and the formation of new neurons in fact forms a crucial part of the learning process [121, 125, 394]. For an overview of this paradigm shift in neuroscience see [300].

Neurons operate in parallel and asynchronously; there is no centralised control mechanism to provide timing information for the network. Individual neurons are relatively autonomous in that they have no direct knowledge or access to the states of other neurons — their output depends only on their own state and the signals which they have received along afferent connections.

Neurons are analog processing units, which is unsurprising as they exist in the real world. They operate on signals which are continuous in both the activation and temporal dimensions. In neural network research, the “state” of a neuron has traditionally been interpreted as the voltage difference across its cell membrane.

Figure 2.2 shows the structure of a biological neural cell. The dendrites are the inputs to the neuron. Signals arrive along them from the outputs of other neurons and from the central nervous system. The axon carries the output signal from the neuron to terminal buttons. The dendrites present inhibitory or excitatory signals to the neuron which have a cumulative effect in stimulating it. If the sum of these input signals exceeds some threshold the neuron will fire, sending an activity spike along the axon. This signal will arrive at the terminal buttons, which will release neurotransmitters into a gap (the synapse) between the button and a dendrite of the target neuron.

The properties of this synaptic gap determine the strength of the signal received by the target neuron, and whether it will excite or inhibit its activation. Although it is possible that physical properties of the dendrite and axon connections might perform some processing of carried signals, it is not believed that this is the case, as they seem to accurately reproduce the input signal. The fastest neurons have switching speeds of 10^{-3} seconds, which is relatively slow when compared to the 10^{-10} switching times of semiconductor transistors [306].

The classic view of information flow in biological neurons has been that signals are propagated unidirectionally along the dendrites from synapses to the soma. It is now known that many neurons also send information backwards, from the axon to the dendrites, and that this feedback can stimulate plasticity changes in the synapse [6, page 30].

Cnidarians are a phylum of animal species that share a common and unique form of nervous system. The nervous system consists of decentralised nerve nets, with no brain. These nerve nets have no dendrites, axons or synapses, instead neuron cells are directly stimulated by other neuron cells that they happen to be in physical contact with. This means that connections are bidirectional and apparently randomly organised. Some creatures are not capable of locating stimulus points, and will react the same way to certain types of stimulus regardless of its point of origin. Since all connections are bidirectional, neurons can receive feedback echoes from activity spikes that they originated or already propagated.

Computational neuroanatomy studies suggest that evolution has optimised the positioning of macro-structures in the brain, their long distance interconnections, and the position of the brain within the body, in order to minimise volume and overall wiring distance [65]. Evolution has discovered structures that are close to optimal — in some direct comparisons between actual neural structures found in biology and corresponding theoretically optimal structures it was found that the volume of the real-world

structures was less than 2.5% larger than the optimal topologies, and that the real-world structure was in the top 0.14% of possible configurations [64]. A connection-cost analysis of the macro-layout of functional areas of the macaque visual cortex showed that the actual layout was in the top one-millionth of all alternative layouts [66]. This is impressive given that three-dimensional packing is known to be an *NP*-complete problem.

Cherniak suggests that neural optimality may have been achieved through the exploitation of physical properties of the world, rather than being explicitly encoded in the genome; the neuroanatomical layout of *C. elegans* can be exactly reproduced by a mechanical model in which each connection of the *C. elegans* nervous system corresponds to a micro-spring, and the system is allowed to fall into vector-mechanical equilibrium [63]. The resulting optimal layout of the model is the best of a total of fourty million possible configurations.

The precise way in which the topology of a brain is constructed is unknown; the human genome contains around 24,000 genes, and the human brain consists of 100 billion (10^{11}) neurons and 100 trillion (10^{14}) synapses [81, page 59]. 96% of the human genome is shared with chimpanzees [459]; the 4% difference is made up of single-nucleotide changes, duplications of existing sequences, and newly inserted or deleted sequences. Since diverging from chimpanzees, humans have gained 689 genes, and lost 86 genes [102]. Transcription factors, which affect the transcription of other genes, are four times more likely to have changed than other genes [157]. The difference in intelligence between chimpanzees and humans must therefore be accounted for by a mere 689 genes, and the morphology and functionality of the whole human brain and body must be accounted for by only 24,000 genes. The total amount of brain-specific DNA in the human genome is estimated to be around 100 megabits (12.5 megabytes) — too little to encode complete information about the location and connectivity of every neuron and synapse [63]. The discrepancy between the large number of neurons and synapses, and small number of genes, suggests that the precise location and connectivity of each neuron can not be encoded directly in the genome, and must instead be represented using a largely compressed encoding (this problem has been termed the “poverty of the genetic code”) [81, page 59], with the genome specifying basic structure, such as the relative positioning of functional centres and the pathways of connectivity between them, rather than the precise topology of individual neurons.

2.2 Biological models

Biological neural networks are complicated, and we still do not fully understand how they function. The Hodgkin-Huxley model, first described in 1952, is the most widely accepted model of neuron action potentials [192]. The model has been refined over time, but some problems still exist. In 2005 Heimburg and Jackson proposed a soliton model, in which computation in neural networks is carried out by waves of sound propagating through the neural membranes. It is claimed that the observed change in action potential, which is usually considered the primary mechanism of computation within a neuron, is actually a secondary effect caused by changes in membrane density and thickness as soliton waves flow across the network [184].

The simple “network of neurons” model does not account for chemical diffusion in which molecules, such as nitric oxide, pass through cell walls unhindered [210, 405]. This allows neurons to influence other neurons that are nearby in 3D space, rather than being directly connected as in traditional models. Although such diffusion has been observed in neural cells, it is not known whether it plays an important part in the functionality of biological neural networks.

Models of neural processing that rely on quantum effects have been proposed [330]. Similarly, it has been proposed that sensory perception, such as the ability to differentiate between different smells, actually relies on quantum entanglement between neighbouring cells, and hence can not be explained or simulated using network models [46, 452].

2.3 Connectivity and resilience

Biological neural networks typically have a huge number of neurons, which are greatly interconnected. A human brain has approximately one hundred billion (10^{11}) neurons, each of which has around ten thousand (10^4) synaptic inputs on average, making a grand total of around a quadrillion (10^{15}) synapses [306]. This large number of neurons and connections makes them highly resilient to damage — typically around 1000 neurons die naturally each day of a human’s life, and yet there is no noticeable change in the day to day behaviour or intellectual capacity of individuals, suggesting that it is composite networks, and not individual neurons or connections, that are responsible for reasoning and behavioural computation.

The connectivity of an individual neuron is determined by its type and location

within the brain. Connections are more likely to exist between neurons that are physically close, so the distribution of connections is locally dense and remotely sparse. The brain also shows elements of compartmentalisation and modularity; it is known that some regions, such as the hippocampus, consist of densely packed, regularly arranged neurons, and that these regions perform distinct modular functions related to higher reasoning, long term memory formation, etc.

The nervous system of the microscopic worm *C. elegans* consists of 302 neurons, and is the only organism for which a complete connectivity diagram exists. The network topology has been classified as a “small-world network”, in which the clustering coefficient is high (directly connected nodes are likely to have neighbour nodes in common), and the “characteristic path length” (the average length of the shortest path between two nodes) is low [475]. This suggests the effect of evolution under real-world constraints has been to maximise connections between physically close nodes where the cost of an edge is low, to minimise connections between distant nodes where the cost of an edge is high, and to simultaneously minimise the average hop distance; i.e. optimising the neural network model towards lower propagation delay and increased computational power. This is a similar hypothesis to the one Sipper proposed when discussing the “average cellular distance” metric for evolved non-uniform cellular automata (see section 6.9).

New techniques are being developed which combine fine mechanical slicing of a brain, high resolution two-dimensional electron microscopy imaging of individual slices, followed by automated machine learning algorithms, to build a three-dimensional connectivity map of entire brain sections [399]. Another new technique called “diffusion spectrum imaging” uses MRI scans to detect water molecules along axons and automatically build three-dimensional connectivity maps of the brains of living creatures (figure 2.3) [169, 170, 400].

There are many cases of people receiving enormous amounts of damage to the brain and showing no ill effects to either their physical control ability or mental reasoning capacity. However, there are other cases in which enormous negative effects have been observed, and yet the brain has shown a remarkable capacity for regeneration. In 1984 Terry Wallis was in a car crash which resulted in severe brain damage. He spent 19 years in a “minimally conscious state”, in which he was technically awake, but unable to speak, move, or communicate in any way. His recovery in 2003 was attributed to massive regeneration of the neural connections in his brain; magnetic resonance imaging scans showed levels of metabolic activity that were significantly

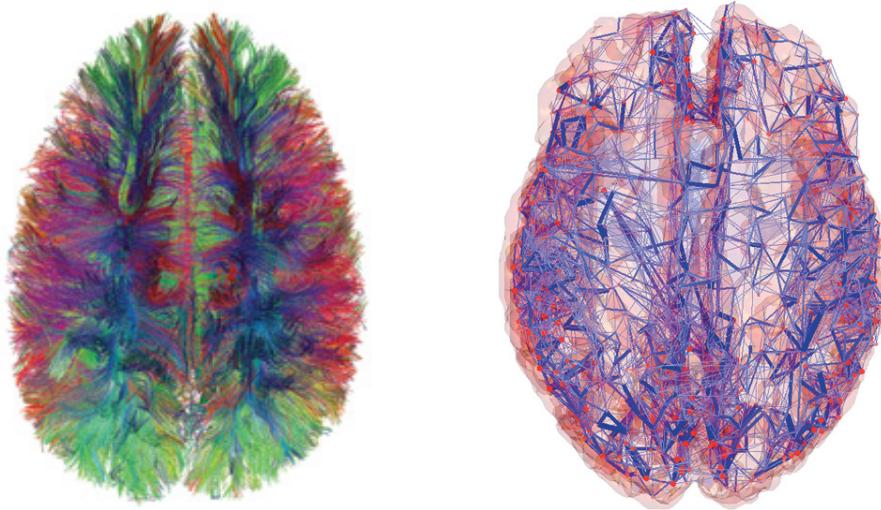


Figure 2.3: Left: Diffusion spectrum imaging is used to gather high resolution data showing connectivity in the brain. Right: Important structural features can be determined by analysing the number and length of connections between different parts of the brain. Bundles of connections between clusters of neurons are visualised here as straight lines, with line thickness weighted by the size of the bundle.

Credit for image: Van J. Wedeen, Patric Hagmann, and Olaf Sporns [169, 170]

higher than other patients with similar brain injuries who had not recovered [470]. He is now able to speak and has some movement ability. Wallis's recovery surprised neuroscientists, who generally believed that recovery from such a prolonged state of damage was unlikely.

Similar observations have been made in other cases where the brain has been impaled, struck with a blunt object, shot, or damaged by burst blood vessels, cancer, or invasive surgery. Long term coma patients have “awoken” and recovered, despite previous scans showing only trace amounts of brain activity [256]. It is not always the case, but quite often the brain displays a remarkable and unexpected capacity for resilience and self-repair [111].

Other studies have shown that the brain when damaged, even though it may not self-repair, can still function relatively well, suggesting that functionality is modular and autonomous, and that communication between different modules, although advantageous, is not strictly necessary.

The left and right hemispheres of the brain communicate over a thick bundle of around 200-250 million nerve fibers known as the corpus callosum. “Split-brain” syndrome, in the corpus callosum is severed, is particularly interesting [148]. Patients are

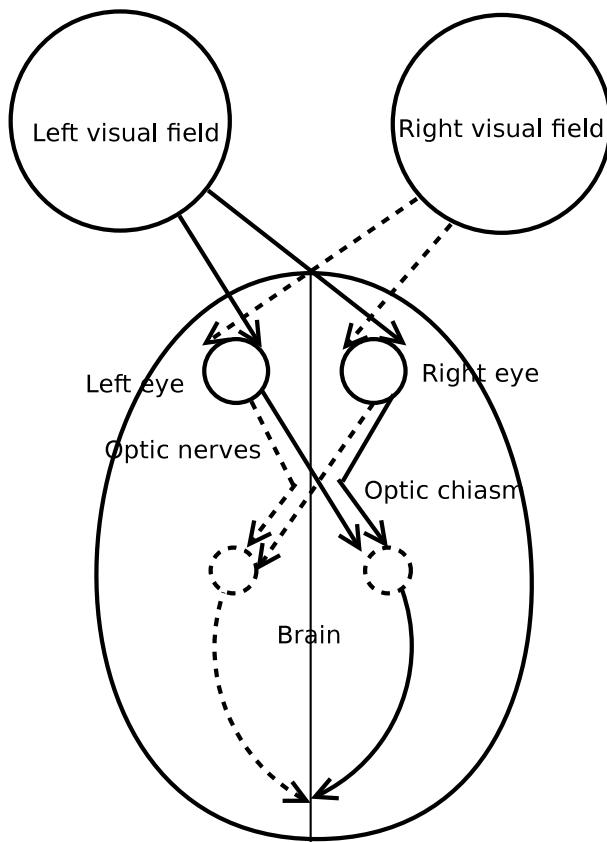


Figure 2.4: *Visual pathways of the human brain. An eye senses portions of both the left and right visual field, providing some amount of redundancy. Each visual field maps onto the opposite brain hemisphere. The bipolar field of vision where the visual fields of the two eyes overlap enables 3D imaging.*

able to see, comprehend and act in a localised manner in which their body appears to be vertically split down the middle. Surgical operations to split the brain are sometimes carried out to cure epilepsy; it has been hypothesised that severing the link prevents epileptic seizures from spreading from one brain hemisphere to the other, and perhaps prevents a seizure in one hemisphere from being amplified by a positive feedback loop.

Due to the process of evolutionary development the left and right sides of the human brain are wired to sensors and muscle on the opposite side of the body, e.g. the left brain hemisphere is connected to the right arm, leg, etc. and vice versa [2, 70]. The eye is slightly different; each eye senses portions of both the left and right visual fields, with the signals from each visual field being combined at a central structure known as the optic chiasm, and then routed to the opposite brain hemisphere (figure 2.4). The region where the fields of vision of the two eyes overlap is known as the binocular field

of vision, and it enables the brain to construct a three dimensional view of the world.

At the neural level, people with split brains are effectively two separate individuals, who together operate a single body. Since there is no direct neural connection between the two hemispheres (they are connected indirectly via the shared brain stem), each side is completely unaware of what the other side is sensing or calculating. For example, a typical right-handed patient, using the right brain hemisphere, will be able to visually locate a specific object in their left visual field, pick it up with their left hand, and operate it in the manner of that object's use, and yet will be unable to name the object, as language and object-to-name mapping skills are located in the left side of the brain. Similarly, written words can only be read if they are in the right visual field, which is connected to the left hemisphere where language processing resides.

The “split-brain” syndrome confirms that the brain is modular. Patients are able to perform complicated tasks, such as walking and playing ball games, that we would have otherwise assumed result from some learning of a sequential action sequence that coordinates muscle signals to both sides of the body from a single position within the brain.

Another interesting associated phenomenon is that when a normal (non-split brain) individual is viewed in an instance, say a photo, their facial expressions are not symmetrical; although this is difficult to comprehend until compared to a digitally composed image of what such a symmetrical face would look like. It can be seen that, whilst one half of the brain may be smiling, the other half will appear only slightly amused, demonstrating the disconnect in functionality and connectivity between the two brain hemispheres.

The story of Abigail and Brittany Hensel also suggests that complex physical actions can result from the operation of two completely separate control systems. The Hensel sisters are conjoined twins, each possessing a head and brain, but with spinal cords that fuse at the pelvis to form a single lower body (figure 2.5). Each brain receives sensory information from only half of the body, vertically split, and can control muscles only on that side of the body. Despite this, the twins can coordinate control of their body to play basketball, cycle, swim, and, since they turned 16, drive a motor vehicle.

One example of brain damage causing a modular loss of function is the inability to form long term memories following the removal or destruction of the hippocampus, as dramatised in the hit movie “Memento”. The hippocampus is a symmetrical structure present in both the left and right sides of the brain, which is involved in the process

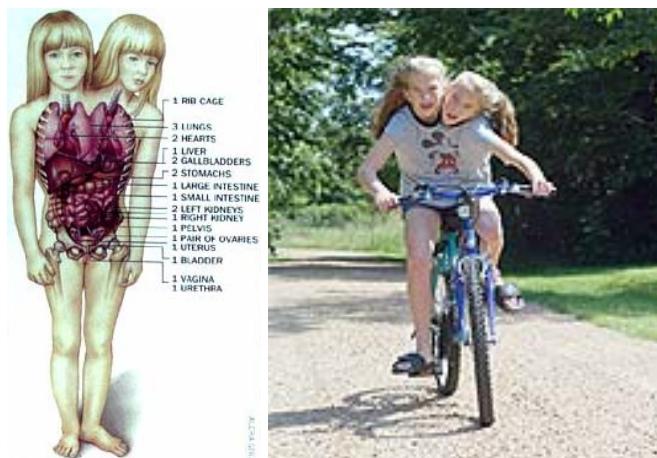


Figure 2.5: The Hensel twins have two completely separate nervous systems sharing a single body, with signals from each brain dividing the body into left and right halves. Despite this, they can carry out actions like cycling with no more conscious thought than others, proving that complex behaviours can arise from the composition of independent control modules at the neural level.

Credit for image: ABC Online

of converting short term memories into long term ones. Damage to either side, where the other is undamaged, leads to minor loss in the ability to remember particular types of sensory information. However, damage to both results in total loss of the ability to remember anything over a period of time. Ability to form short term memories is unaffected.

The loss of long term memory effect was first discovered in 1953 following surgery on the anonymous patient “H.M.” [387]; research on the same subject persisted until his death in 2008, upon which his identity was revealed as Henry Gustav Molaison [56, 75]. The effect suggests that memory function is resilient and distributed across brain structures, it seems likely, given the catastrophic results of a double failure, that when one side of the hippocampus fails the other somehow compensates. In recent years work has begun on a prosthetic neural interface that could be used to replace the hippocampus and restore memory function [25]. Researchers have been able to model the signal transformation function performed by the hippocampus with 95% accuracy, despite not understanding how it actually works.

The flip side of this is that human behaviour is produced by extremely complex interactions between large numbers of neurons. A small amount of damage spread across a large area of the brain will often result in no noticeable changes in the person,

but localised damage of an important area can result in loss of functionality and change in personality. This is well illustrated in the classic case of Phineas Gage, a 25 year old railroad foreman, who in 1848 had a 91.5cm long, 3.2cm diameter, 6.4kg iron tamping rod blown through his head in a freak accident [176, 318].

Gage was busy packing explosives with the tamping rod when they detonated, propelling the rod upwards through his head. The rod entered just below the left cheekbone, passed vertically through the brain, and exited via the top of the skull, landing 30 metres away. Surprisingly, Gage survived, and regained consciousness only minutes later. After some time of recuperation, Gage recovered physically. Unlike many victims of such severe brain damage, his powers of movement and comprehension appeared to be unaffected. However, Gage's doctor and friends noted that his personality had changed radically.

Before the accident, Gage was known as a decent, stable person, a smart businessman, who was adept at methodically formulating and carrying out plans. After the accident, he became abusive and impatient towards others, and would constantly swear. He developed an impulsiveness and lack of attention that left him unable to plan and act on those plans as he once did. With his new personality Gage's employer was unwilling to return him to a position of responsibility; in subsequent years Gage held down a series of manual labour positions, but was never again employed in any profession that required planning or managerial skills. Eleven years after the accident, Gage developed epilepsy, and he died several months later.

Gage's skull is on display at Harvard's Countway Library of Medicine. Examination shows a large exit hole sized $5\text{cm} \times 9\text{cm}$; the probable path through the brain has been computed with the help of computer aided tomography (CAT) scans (figure 2.6). A further analysis concluded that, despite the severity of the accident, damage had been restricted to the left frontal lobe [347], an area associated with impulse control, problem solving, and the carrying out of behaviour.

2.4 Pattern generation for muscle control

The control of movement in animals and humans has been studied by physiologists and neuroscientists throughout the 20th century. Their experiments have shown that most cyclic motions of the body are generated by parts of the nervous system known as “central pattern generators” [195, 270, 279, 307]. These consist of collections of neurons which generate rhythmically recurring patterns of output signals. It is these

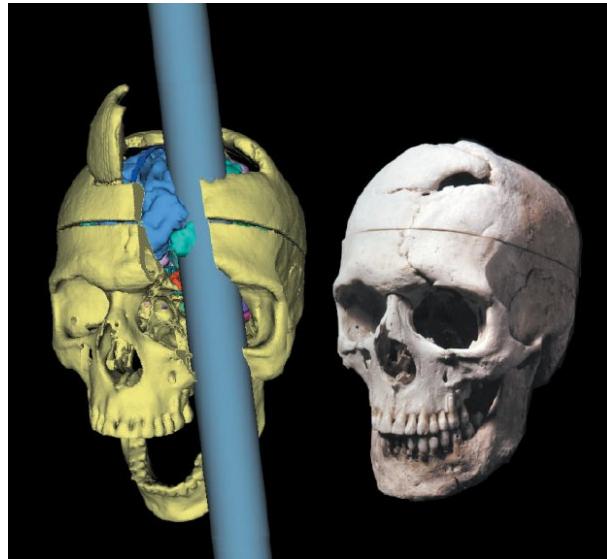


Figure 2.6: The brain injury to Phineas Gage, reconstructed from CAT scans (left), and the actual skull (right). The computer generated image includes the tamping rod which was projected through his skull. Brain damage was localised to the left frontal lobe, resulting in a complete personality change. Other brain functions were unaffected.

Credit for image: Peter Ratiu and Ion-Florin Talos [347]

signals which are presented, via motor neurons, to the muscle actuators to drive bodies through movement, such as walking, swimming and jumping.

The dynamics of a central pattern generator act as a limit cycle attractor, causing its neurons to produce activity patterns which oscillate, propelling the network along its attractor path, and eventually returning it to its original state from which the cycle begins anew. If the neurons of a pattern generator lose synchronisation the dynamics of the network ensure that they will fall back into a similar, but not necessarily identical, rhythmic pattern.

It has been theorised that every body part that makes cyclic movements is controlled by an individual central pattern generator [97]. Experimental evidence has shown this to be the case for certain creatures; in particular, due to the unique biology of the lamprey eel, neurophysiologists have been able to map the neurons responsible for pattern generation and synthesise their behaviour, showing that central pattern generators can reproduce natural movements of the creature [166, 171]. Central pattern generators have been shown to produce more stable gaits when compared to reflex based controllers on a robot walking task [127]. A central pattern generator has been used to synchronise the motion of two independent neural networks to drive

biped robot walking [461]; a staged genetic algorithm was later used to add modulating sensory input that enabled the neural networks to adapt at runtime to variation in morphology.

Ascribing simple cyclic movements of an individual muscle or muscle group to a neural pattern generator sounds reasonable, but are these pattern generators interconnected, and if so, does the connectivity form a top-down control hierarchy, or a loosely connected distributed control system? It is likely that the truth is a mixture of both; human behaviours such as walking are initiated and modulated by higher level control centres, but individual actions and patterns of muscle activation are most likely learnt by adaptable individual pattern generators.

It has been proposed that a hierarchical control system is built by linking generators that perform coordinated behaviours, or by introducing a “parent” central pattern generator which activates and modulates those behaviours [228, 270]. For example, generators for different muscles in the same limb may be connected to a limb-wide central pattern generator which orchestrates the lower level ones into producing coordinated limb movements. This limb-wide generator may in turn be connected to other pattern generators in other body parts, so that the movement of multiple limbs may in turn be coordinated.

Mackay and Lyons suggest that a variety of central pattern generators may exist for each joint or muscle group, and that these generators are parameterised by electrical and chemical stimulus. The interactions between generators coordinate movements which are somewhat hard-coded within the creatures morphology; thus higher level motor learning is reduced to the task of combining and switching between different central pattern generators, rather than learning patterns from scratch [270]. Patterns generators do have some degree of plasticity, as cats with transected spinal cords have displayed progress in stepping and standing with different training regimes (the cats were trained to step or stand, but not both). Progress was only observed for the specific trained action (step or stand), and not the untrained action, indicating that the function of the central pattern generator had changed in response to the spinal injury and training [270].

In order to study central pattern generators in more depth, biological neural networks have been created in-vitro. Syed reconstructed the central pattern generator of respiratory rhythm in the mollusk *Lymnaea* [426]. An analysis of its structure showed that respiratory rhythm in *Lymnaea* is generated by a network consisting of only three neurons. Later research has identified multi-function controlling neurons which mod-

ulate rhythm generation, and coordinate sensory input and motor control [175].

2.5 Sensory feedback

Proprioception is the sensing of the internal state of the body, which is communicated to the brain, forming an internal action-sensation feedback loop. This enables the nervous system to detect the current position of limbs, and forces being exerted by muscles, and is important in providing timing information to central pattern generators. Exteroception is the sensing of the external world. Such feedback could provide valuable information about position and orientation of the body within the world, contacts between the body and other objects, and visual information on more distant objects.

Sensory feedback has been observed to activate and inhibit, though not generate, patterns of activity. Experiments on deafferented animals, where the spine or nerves have been severed to prevent sensory feedback, have shown that similar activation patterns will still be generated, but the timing of body motion and patterns becomes desynchronised, and coordinated movements become erratic [165]. Transection studies, in which the forebrain is removed but lower levels left intact, have shown that low-level behavioural mechanisms can operate independently of higher level control systems, although they lack inhibition and will activate in inappropriate contexts [344].

Feedback can drive certain generators into switching, with smooth and stable transitions, between the production of alternate patterns. One example of this is in gait generation for walking. Not only must pattern generators produce coordinated cyclic activation of muscle groups in each leg, they must also adapt this activation to cope with transitioning between dynamically unstable gaits whilst keeping the creature stable and upright.

One way in which this could be done is to connect the pattern generators from each leg to a series of gait pattern generators. As the speed changes the current gait gradually becomes unstable, its gait pattern generator is driven out of its attractor cycle and becomes inactive, switching smoothly to activate an adjacent gait pattern generator. The output activity patterns of adjacent gait generators would be similar enough that the body could switch between them without causing the creature to fall.

There is still some dispute over the exact role of sensory feedback, reflexes, and central pattern generators in living creatures. In the animal world, moving the tail of a paralysed dogfish from side to side will stimulate its motor neurons into producing activity at the frequency of the imposed oscillation. This means that feedback is di-

rectly responsible for generating activity patterns, possibly through reflexes and not pattern generators. A reflex based controller has been used to successfully control a walking biped [327], showing that central pattern generators are not strictly necessary for complex control tasks.

2.6 Brain computer interfaces

Interfacing biological neural systems to silicon chips allows both for real-time analysis and reverse engineering of biological models, and the actual use of biological networks in control and computation tasks. The electronic circuitry connected to the brain is known as a “brain-computer interface” (BCI).

Biological pattern generators are adaptable and can be trained to perform specific behaviours. In 2000 Reger *et al.* carried out a set of experiments in which the brain and central nervous system of a lamprey sea eel were removed and connected via a neural-silicon interface to a two wheeled robot body (figure 2.7) [245, 351].

The lamprey brain could successfully sense light and control the motorised wheels of the robot body. The response of the cyborg to varying light stimulus was analysed and reduced to a set of differential equations, showing that the lamprey displayed light following behaviour. This could be changed to light avoidance behaviour by relocating the electrodes in the neural tissue. The test system enabled neural spike trains to be sent directly from a computer interface to the lamprey brain. This was used to validate the model by inputting precise data points and observing the output motor electrode activation. It was then shown that the brain could adapt to an increase in light sensor sensitivity.

Since these behaviours can be trained into the neural network of the lamprey, and we know that the lamprey neural system can be synthesised by artificial central pattern generators, we have reason to believe that a robot control system utilising central pattern generators will be capable of displaying both movement, learning and adaptation, and other more complex behaviours.

In 2003 Fromherz reported growing biological neurons on silicon chips (figure 2.8) [140]. The neuron cells are encouraged to attach directly to transistors to form a neuron-silicon junction so that signals can pass between biological neural networks and on-chip silicon transistor based networks. This enabled researchers to perform non-invasive monitoring of signals passing through biological networks, by encouraging their growth over, and attachment to, silicon probes. Signals could be input to the biological net-



Figure 2.7: *The sea lamprey (left)* is a common subject of neurological research due to its easy to manipulate nervous system. The brain has been used to control a Khepera mobile robot (right)

Credit for image: Center of Marine Biotechnology, University of Maryland and Institute for Theoretical Computer Science, Graz University of Technology [245]

work by stimulating the neural cells directly with electrical charges [211, 226].

In 2004 Xu *et al.* directly interfaced silicon chips to the brains of living rats (figure 2.9). The system consists of a rat backpack containing a radio transceiver, microprocessor, brain interface circuitry, and a battery. The backpack is completely self contained, allowing unrestricted movement of the rat in 3D space. The radio interface links the microprocessor to a PC, and the signal is digitally encoded to preserve integrity.

The brain interface circuitry allows multi-channel stimulus of several brain regions associated with sensory perception from the whiskers and pleasure. An operator can remotely control the rat using a standard laptop from a distance of up to 300 metres. Controls consist of “move left”, which stimulates neurons usually activated by contact with the right whiskers, “move right”, which performs the same operation on the left whisker neurons, and “reward”, which stimulates the pleasure centres to reward the rat for an action it just carried out, or to encourage the rat to continue whatever it is currently doing (i.e. turning, or moving in some specific direction).

Real-time remote control of animals has some interesting applications, the most significant being military based, such as espionage (e.g. a rat equipped with a miniature microphone, or bird with video camera), weaponry (animals used to deploy bombs), and surveillance (dolphins deployed to guard sea ports from enemy divers). Other more peaceful operations are also envisaged, such as search and rescue within burning

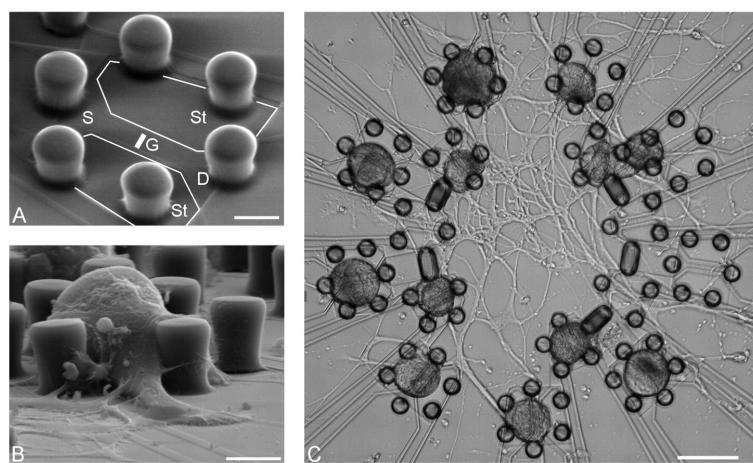


Figure 2.8: *Snail neurons interfaced directly to a silicon chip. Neurons are encouraged to grow within a silicon junction consisting of six contact points.*

Credit for image: Peter Fromherz, Max Planck Institute of Biochemistry [490]

buildings, cities devastated by earthquakes, and other disaster zones.

In 2005 DeMarse cultured rat neurons to control a PC based 3D flight simulator [100]. 25,000 embryonic rat neurons were cultured across a grid of 60 electrodes which can both sense and stimulate neuron activity (figure 2.10). Although 60 electrodes were present, only 2 were actually used in the control experiment. These 2 electrodes were fed with the pitch/roll feedback from the flight simulator, and their response measured. High or low frequency stimulus was then used to adapt the neural cells around the electrodes into either producing a greater or lesser response. In essence, the network was trained to calculate error functions for the pitch and roll variables.

Once trained, a process which took only minutes, the network could successfully control plane flight with only minor (less than 10% degrees) deviation from the perfect response. Previous research by the same team includes the interfacing of rat neurons to a simulated virtual creature with both sensory and control pathways; however, the creature was not trained to perform any specific task [101].

The sensing and decoding of motor control neurons is an active research area which has huge implications for people suffering from paralysis. Recent developments in this area include the 2006 human implantation of a “BrainGate” chip, designed to monitor and decode hundreds of neurons simultaneously and communicate activity back to a wearable computer [191]. The prototype chip has 100 electrodes, and has been successfully used by the implanted subject to control the movements of both a

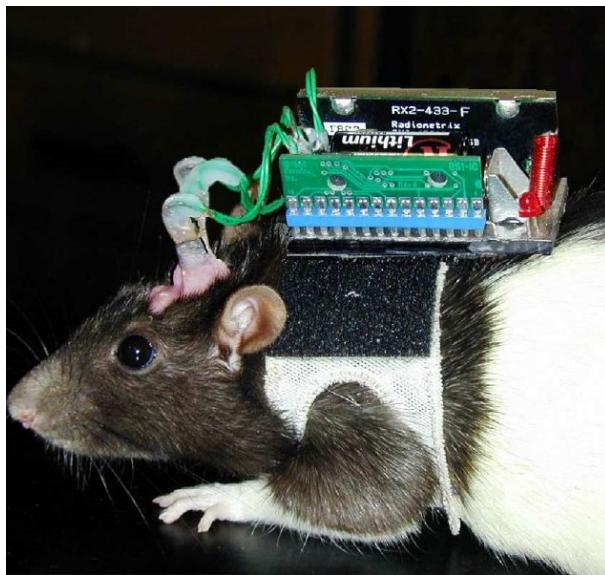


Figure 2.9: *A silicon chip directly interfaced with the brain of a living rat. The chip is driven by a PC connected to a radio base station into providing multi-channel stimulus of neurons associated with left and right whisker sensation, and the pleasure centre of the brain.*

Credit for image: S. Talwar, State University of New York [484]

robotic arm and the cursor of a computer user interface.

2.7 Summary

This chapter explored biological neural networks. Biological neural networks are found in the brains and nervous systems of living creatures, and consist of neuron cells and connections between those cells. Biological networks are modular, complex and adaptable, and resistant to damage and degradation. Mathematical models of these networks have been developed. The function of an individual neuron has been modelled, and the model has been verified to be reasonably accurate. Despite this, the mechanism of how individual neurons form larger computational networks is unknown. Some creatures with very small neural networks have been modelled, but there are no successful models of larger networks such as complete mammal brains.

This thesis explores the effect of quantisation on the performance of evolved neural networks that control simulated systems and virtual creatures. Evolved controllers must generate cyclic patterns of activity in order to drive behaviours such as locomotion. The “central pattern generators” of real nervous systems provide a biological

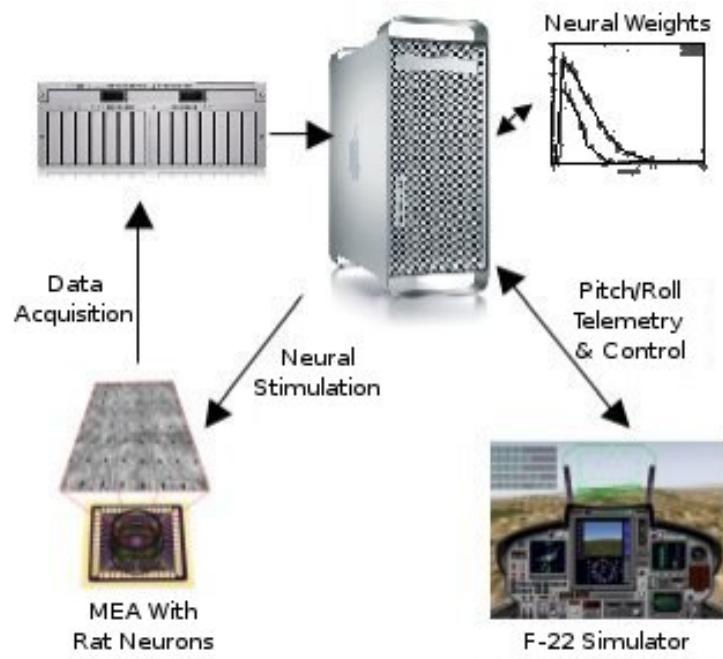


Figure 2.10: *Rat embryonic neurons are interfaced to a microelectrode array (MEA), which allows simultaneous sensing and stimulus of neuron activity at multiple points within the biological network. The electrode array is connected to a virtual world simulation via a PC.*

Credit for image: T. DeMarse, University of Florida [100]

basis for artificial neural behaviour. The role of sensory feedback is also important in stimulating and modulating cycles of neural behaviour. It would greatly help the implementation of virtual and artificial creatures and their controllers if we understood how biological creatures functioned.

This chapter has also looked at the topic of Brain computer interfaces. This is an interesting research area as it brings together two topics that are relevant to this thesis — biological neural networks and VLSI analog/digital hardware. It is likely that further research in this area will produce advances in the understanding of neural coding in biological creatures, which is a topic of great interest to implementers of virtual creatures.

Chapter 3

Synthetic neural networks

In the last chapter we looked at biological neural networks. This chapter will describe the theory and operation behind synthetic neural networks, which attempt to recreate and simulate the networks that we see in nature. The study of synthetic neural networks is an important aspect of this thesis — synthetic neural networks have been used by other researchers for robotic control, including control of evolved creatures. The research carried out as part of this work involves using genetic algorithms to create synthetic neural networks that can control simulated robots, and then comparing the performance of neural models with different levels of quantisation on some typical dynamic robot control tasks.

Synthetic neural networks have been implemented in a variety of technologies, such as software running on generic processors, analog and digital VLSI, programmable logic circuits, cellular automata, and custom processors. One of the aims of this thesis is to explore whether quantised neural networks, which can be implemented with fewer hardware resources than continuous networks, can be used for robot control tasks, and if so, how degraded their performance will be. Prior implementations of neural network processing systems are interesting from this perspective, as the designers of these systems will have already attempted to optimise their systems to maximise neurons simulated whilst minimising computational resources. Various neuron models and network topologies will be introduced, as well as some popular methods for training a network to perform generic pattern matching and classification of input data patterns.

The principal motives behind synthetic neural network research are to gain further insight into the function of biological networks and to create something that is actually useful. Commercial applications of neural networks were non-existent until the 1990s,

mainly due to lack of affordable computing power. Neural networks have now become the favoured architecture for pattern recognition, leading to a variety of applications in machine vision, fraud detection, speech recognition, and many other tasks which require robust and reliable generalisations to be made from noisy input data.

A neural network is modelled as a directed graph of neurons. Each neuron operates with some strictly defined mathematical functions which transform its input signal values into internal activity, and its internal activity into an outgoing signal value. Connections between neurons are often weighted, so that the effect of a signal is amplified or reduced, and either the connection, or the source neuron, is said to be excitatory or inhibitory, and so will act to either increase or decrease the activity of the receiving neuron. The input signals to an individual neuron are often collected together by summing the weighted signal values. A neuron produces a single output signal which fans out to many receiving neurons, each of which will receive an identical copy of the signal, though some models may introduce a small phase shift to simulate synaptic delay.

Synthetic neural networks were first studied by McCulloch and Pitts in 1943 [286]. They described a network structure, where input nodes reproduce externally sensed signals, feed-forward edges connect internal nodes, which perform computation, the results of which are ultimately presented on output nodes. They argued from biological principles that a simple summing model with a binary threshold was realistic, and showed that, under this model, there would always be some network to implement any given logical expression.

Their neuron model had binary inputs and outputs, the activation function was the Heaviside step function (see section 3.5), and inputs were not weighted but could be excitatory or inhibitory. The threshold level could be varied to change the logic function (e.g. with two inputs, a threshold of 1 produces an OR gate, because $1 + x \geq 1$, or a threshold of 2 makes an AND gate, because $1 + 1 \geq 2$). The inhibitory connections allow the reproduction of signal inversion necessary for the NOT, NAND, and NOR functions. A single active inhibition signal forces the output of the neuron to 0, regardless of the other inputs; this is known as “absolute inhibition” [365].

In 1958 Rosenblatt introduced the “perceptron” model [366]. The classical perceptron, as defined by Rosenblatt, differs from the McCulloch-Pitts neuron only in that it has edges weighted with signed real-values, and uses “relative inhibition”, where a neuron’s activity is reduced by an amount proportional to the weighted sum of its active inhibition inputs. Rosenblatt defined a precise feed-forward network topology

consisting of binary sensors projecting signals to an input layer through deterministic connections. The input layer is in turn randomly connected to a perceptron layer, which is itself randomly connected to an output layer. The input and output layers merely reproduce signals and do not perform any computation, so this is actually a single layer topology.

The first gradient descent training algorithm, “least mean squares”, was published by Widrow and Hoff in 1960 [478]. At the time, many ambitious predictions were being made regarding connectionist computing. In 1957 Simon claimed that within 10 years a computer would become world chess champion, and a major new mathematical function would be proved by an automated system [369]. It was widely believed amongst the AI community that within 50 years computer intelligence would exceed that of humans, and one of the great dreams of science fiction would be realised.

Fifty years on, and our most optimistic commentators still claim that human levels of artificial intelligence will be achieved within the next 50 years [312]. It has been realised that the problems being posed were much harder than we first thought. Even the challenge of beating a world chess champion, a task seemingly ideal for repetitive computation, has proven to be a formidable one, that has only recently been achieved [208]. In sensing and control tasks the synthetic neural networks of today are comparable to the most basic insects in their complexity and observable behaviour.

It was not until 1997 that Kistler directly compared the threshold logic of McCulloch and Pitts to the widely accepted Hodgkin-Huxley model of a biological neural network, showing that the single variable threshold gate could correctly predict 90% of the spikes generated by the more complex Hodgkin-Huxley model [233].

In 2005 the Hodgkin-Huxley model, used as the biological justification of the McCulloch and Pitts action potential model, was disputed by Heimburg and Jackson, who proposed that computation in biological neural networks is carried out by solitons — propagating waves of sound and pressure — and that changes in action potential are only a secondary effect of changes in membrane density and thickness [184]. If this is the case, then it would undermine the biological basis of synthetic neural networks, and therefore much of the work done to-date on them. Research on the soliton model is ongoing.

3.1 Similarities to biological networks

In many ways synthetic neural networks are similar to the biological ones which inspired them. Neurons act in a way which models their biological counterparts, providing a function with an arbitrary number of inputs and a single output. They operate in parallel and asynchronously, and higher order ones are capable of the complex behaviours demonstrated by biological neurons.

The synaptic potential which alters the strength of a signal passing between a terminal button and a dendrite is emulated by multiplying transmitted signals by a connection weight. Synapses can be excitatory or inhibitory, which is analogous to multiplication by a positive or negative weight. In networks where edges can carry signed (positive or negative) values, the polarity of a signal can be inverted by multiplying by -1 .

In synthetic networks the connection weights are usually constant at runtime but varied using some offline training algorithm, however, most online learning models will dynamically alter the weights. Individual neurons and synapses may have different time constants, thresholds and delays, although in many algorithms these are constant and only the weights and connectivity are varied. In biological networks, the strength of synapses is varied, obviously while the creature is living, though it could be said that offline training was performed in the case of the neurons interfaced to silicon transistors presented in section 2.6.

Biological networks tend to be on a much larger scale than synthetic ones, containing more neurons and more interconnections between them. This can be seen as a limitation more of our design skills than of technology, as modern vector processors are quite capable of simulating networks with many millions of neurons and connections in real-time.

Simulations on digital computers are inherently digital, whilst biological networks are continuous, but it is not clear that this makes any difference. The quantisation of temporal and electrical continuity approximates reality in that a continuous system allows states to become so similar they can not be separated. Since neurons (and all other computing elements) categorise input signals into different states as part of their function, continuity is perhaps not a necessary prerequisite for complex behaviours.

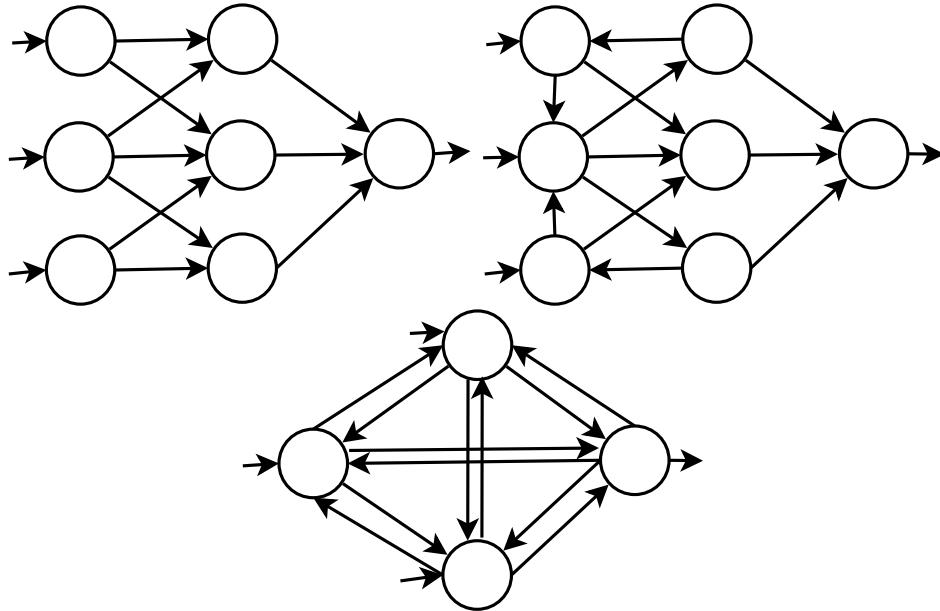


Figure 3.1: *Various topologies of a multiple input, single output, network — feed-forward (top left), recurrent (top right), and fully connected (bottom).*

3.2 Connectivity

The connectivity of synthetic neural networks can be represented by a directed graph. Traditional models use a feed-forward topology, in which signals always travel from input nodes towards output nodes, and there are no cycles. This model became popular as it is easy to analyse mathematically, and yet is still able to approximate any continuous function. Gradient based training methods, like backpropagation, can be used to adjust connection weights to better approximate the evaluation function.

Recurrent neural networks allow the graph topology to contain cycles [329]. These networks are difficult to train with gradient descent algorithms since there is no obvious way to determine how much each node contributes to the final output value. The usual technique of training such a network involves “unrolling” the network to create an almost equivalent feed-forward topology, training that, and back-annotating the weights to the recurrent network [329].

Many recurrent network graphs have arbitrary connectivity. It is possible to shape this connectivity into regular topologies such as one-dimensional rings, two-dimensional tori, or three-dimensional grids. Geometrically regular topologies are unusual in neural network research, but common in the related area of cellular automata.

A network in which the output of every node is connected to every other node is

known as “fully connected”. Again, these type of networks are hard to train with traditional techniques, but are commonly used with techniques, such as genetic algorithms, which treat the network as a black box rather than attempt to analyse its internal dynamics. Connection weights are allowed to vary within some range that passes from positive to negative. Values that tend towards zero will effectively prevent communication between two nodes, which is functionally equivalent to removing the connection, so fully connected networks with varying weights can simulate any other topology.

Experiments have shown that the number of fully connected neurons required to successfully control a dynamic task, like robot walking, can be as low as 10 [355], or as high as 100 [349, 350]. The number of nodes required depends on the neuron model, the connectivity, and the task at hand. Connectivity affects computational ability, since each connection represents another opportunity to carry out a multiplication and communicate data, and so less well connected networks may require more neurons to perform the same task.

3.3 State and signal coding

In biological neural networks signalling is characterised by the temporal “spike train” of a neuron firing. Each spike is a discrete event; no information is conveyed by its amplitude or duration. After firing, a neuron goes through a refractory period in which the generation of subsequent spikes is repressed. Figure 3.2 shows biological spike trains recorded from 30 neurons in a monkey visual cortex over a 4 second period. As can be seen, spikes are not as numerous as may be expected, with a switching frequency several orders of magnitude slower than modern transistors’.

Although we do not understand exactly how information is encoded within the spikes, several mechanisms have been proposed, along with biological justifications for each [155]:

Rate Information is encoded by the mean firing rate (frequency) of a neuron, or group of neurons. This has been discredited by experiments that show computation occurs faster than rate coding would allow. Accurately determining the firing rate requires monitoring and averaging the signal over some period of time. Experiments have shown that a fly can react to external stimulus and change flight direction within 30ms, which is only enough time to generate a single spike. Experiments on monkeys have demonstrated cortical computations that can com-

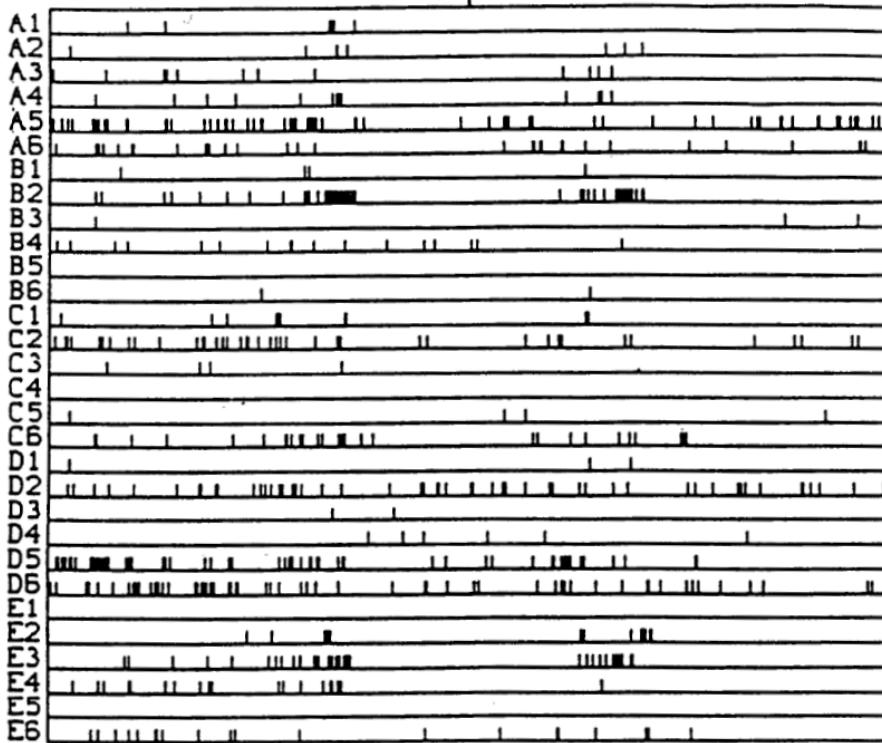


Figure 3.2: *Spike trains from 30 neurons in the monkey visual cortex recorded over a 4 second period. Spikes are not as frequent or numerous as might be expected; there are fewer than ten spikes in parallel over 100ms, and the distance between subsequent spikes is usually over 10ms, in this case stretching to hundreds of ms. Each spike takes 1ms, and is followed by a 10ms refractory period.*

Credit for image: Krüger and Aiple [249]

plete in 20ms, despite neuron firing rates being less than 100Hz [269].

Time-to-first-spike Information is encoded in the temporal distance between the first and subsequent spikes.

Phase Information is encoded in the time between a spike and some periodic signal, such as a reference oscillator.

Correlation and synchronicity Information is encoded by groups of neurons firing together, or firing in specific orders, or in the temporal distance between these events.

In the physical world input and output signals must be temporally continuous and real-valued. Synthetic neural networks can use spikes, floating-point, or multi-value representations internally. Signals from the real world must be translated before being

presented to the network as inputs, and outputs must be translated to real-values before being sent to motors.

If spikes are used, then input signals must be translated into spike trains. If the network uses floating-point values then inputs need only be scaled and translated. For discrete multi-value networks, input signals must be quantised.

Analog neural network implementations, such as custom ASICs, can reliably reproduce biological neural behaviour, and hence, depending on the way in which the system is designed, the information coding of the spikes may have the same interpretation. Synthetic digital spiking neurons may also code information the same way.

The way in which information is represented and encoded in a signal affects how the signal is interpreted. In simulation of continuous networks floating-point values are used for signalling between neurons. The floating-point value can be either interpreted as encoding the frequency of a biological spike train (“rate coding”), or encoding the intended real-value which the equivalent biological spike train would produce if we knew how to decode it precisely (this assumes that biological neurons are actually attempting to communicate real-values, and that spiking is just a low-level way of doing so).

The widespread use of rate coding is partly historical, and comes from a time when it was believed that frequency was the only information carrying component in biological spike trains. The use of floating-point values was also more convenient for gradient descent based weight training. In multi-value networks quantised values represent non-overlapping regions of the signal space, and otherwise are assumed to represent some real-value within that space.

It has been shown that the presence and coding of input signals can have a severe effect on the dynamics of a network. Studies on asynchronous cellular automata have shown that perturbations caused by temporally continuous input signals from the environment cause the formation of large scale regular spatial structures which display long range correlation between cell states, and that these structures are stable despite being continuously perturbed [489].

3.3.1 Example codings

In 1994 Salapura used a delta signal encoding to create space efficient neural networks with bit-stream arithmetic [374]. The “delta encoded binary sequence” signal modulation encodes a real-value number in the range $[-1, 1]$ as the proportional of zeroes and

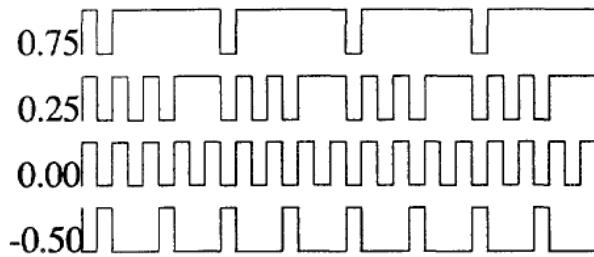


Figure 3.3: *Delta encoded binary sequences of various real-values, where the number of zeroes and ones is proportional to the value being encoded*

Credit for image: Valentina Salapura [373]

ones to output (figure 3.3). No successful use of this coding was reported.

In 1998 Korkin presented the “spike interval information coding” (SIIC) convolution function, which converts a binary sequence into a time dependent real-value output [244]. It is derived from an neuroscience algorithm published by Rieke in 1997 [362]. The convolution function computes the dot product of the last bits seen in the binary stream along with the convolution filter, which is a vector of integers of some pre-determined length. In 1999 the filter values were evolved using a genetic algorithm, and the result was shown to improve accuracy over Rieke’s algorithm by almost 100%, whilst reducing the bitstream look-ahead from 48 to 20 bits [87].

The inverse operation, that of converting a time dependent integer value into a bit-stream, is carried out with the “Hough spiker algorithm” (HSA) [207]. This deconvolution algorithm relies on keeping a running total of the bit changes seen, and inferring how the original input bitstream must have varied in order for the convolution filter to produce the observed output bitstream. See figure 3.4 for a example of how reliably a continuous analog signal is reproduced after being encoded with SIIC and decoded with HSA.

In 2001 Floreano used spiking neural networks to control a two wheeled robot [135]. Inputs to the network came from a 16 pixel video camera. The intensity of each pixel was convolved with a Laplace filter to extract contrast from the vertically lined environment and then scaled to $[0, 1]$ and used as a probability of the input neuron firing. Hence information was coded in the frequency and across neighbouring neurons, which would have values that are somewhat dependent on each other, as they face an almost identical direction. The output value to the motors was not rate coded, but was the difference in the spike count between two motor neurons, so that increased firing on either would move the vehicle forwards or backwards respectively.

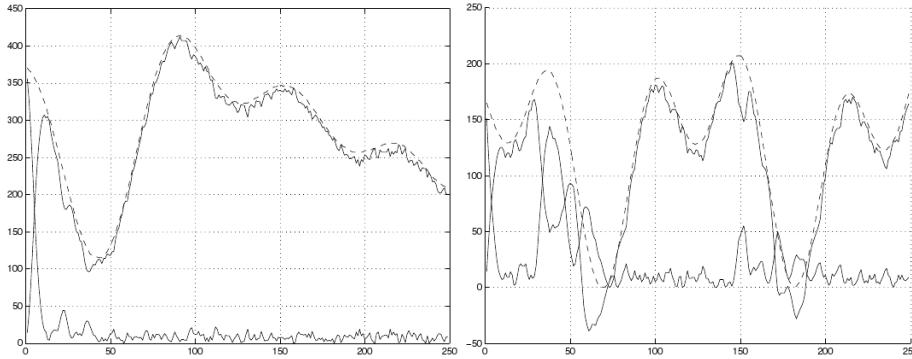


Figure 3.4: A simple test for a coding scheme is to encode and then decode an analog signal, and see how the resulting signal varies from the original. Here, the original signal is a dashed line, and the decoded-encoded signal continuous. The line at the bottom shows the absolute error. SIIC/HSA can reliably reproduce large waveforms which change slowly with respect to the time period of the convolution function (left), but large errors can occur otherwise (right).

Credit for image: Hugo de Garis [93, 207]

In 2002 Floreano extended the above research to a mobile robot running on an 8-bit microprocessor with three infra-red (IR) sensors [134]. Each IR sensor reported a continuous value which was then linearly scaled to one of eight possible discrete states. These eight states were then reduced to a 3-bit value which was coded so as to preserve the property that the number of ones always increases as the sensed value increases. The use of this coding is quite interesting, as most research involving spiking networks uses rate coding of input values, which increases the firing frequency of a single neuron, whereas here higher values directly correspond to more neurons spiking in parallel. The sensors provided new input values every 28ms. This allowed the network, with a cycle period of 2ms, to be updated many times between input activity in order to process internal spikes, so it was not possible for inputs to generate many spikes and saturate the network. The coding of the outputs was the same as the above work in 2001.

In 2003 Schrauwen released a new spike train encoding scheme, “Bens [*sic*] spiker algorithm” (BSA) [383]. Reconstruction of the signal is carried out by a “finite impulse response” (FIR) filter. BSA was shown to have a higher signal to noise ratio than HSA, and more accurately reproduced the input signal.

In 2003 Mandik used the “Framsticks” 3D simulation and evolution environment to evolve and compare a variety of structures of different neural systems [275]. As

a neuroethologist, Mandik was interested in the representation of information within the evolved nervous systems, essentially arguing that information must be encoded in some internal representation, otherwise there would be no way to perform useful computation on it. Mandik argues that delayed copies of both sensory signals and efferent outputs positively contribute to the success of evolved control systems, and that in systems where there is a causal relationship between events and the internal neural representation of an event, and where the representation carries the same form as the original event (in a similar way to a boot print in mud sharing the form of the boot), then these signals encode a representations of the world. Mandik argues that this is in contrast to anti-representationalists such as Beer, who reject that such simple neural networks contain any internal representation of the world [17]. Beer argues that not only is there no representation stored in the state of any given individual node within a simple network, but also that there is no distributed internal representation — that appropriate behaviour and actions are simply the result of current circumstance.

3.4 Models of single neuron dynamics

The chemical reactions that underlie biological neural networks are complicated. At the most accurate level of analysis simulation could be carried out at the level of individual molecules, but this would be computationally prohibitive for simulations of actual networks. In order to simulate large networks mathematical models of the dynamical behaviour of single neurons have been created, at varying levels of abstraction, to allow simulations at different levels of detail, and with different levels of computational power [189].

The Hodgkin-Huxley model of neuron action potential from biology is usually reduced to a single variable differential equation [233]. This equation can be split into separate components computing an input function to gather together incoming signals, an activation function to compute a internal state, and an output function to compute a signal to be sent to other neurons.

3.4.1 Input function

The input function amalgamates the values received on each of a neuron's input connections. Most neural models associate a weight value with each incoming connection, and compute the weighted sum input function $\sum_{j=1}^n W_j I_j$. In effect, this function cal-

culates the dot product $W \cdot I$ between the vector of weights and inputs.

Threshold neurons often have an internal bias value, which shifts the threshold left or right (without a bias the switching threshold would be $x = 0$). For convenience, and faster processing on vector hardware, this bias is often incorporated into the weight vector, with the corresponding input being set to 1 or -1.

3.4.2 Activation function

The activation function defines how the stimulus value calculated by the input function affects the neuron state, or in biological terms, its action potential. Some neural models, such as the sigmoid neuron, have no internal state, in which case the activity function is the identity function. Other common models, such as those from Beer, Taga and Ekeberg (see section 3.4.5), have an internal state, and the activation function specifies how it changes using first, second or third order differential equations respectively.

3.4.3 Output function

The output function (also known as the transfer function) transforms the current neuron state into an output value which is sent to other neurons. The output function is usually restricted to being non-linear — if it were linear, the effects of multiple weighted layers could be reduced to a single layer by multiplying out the weights, and hence the network as a whole would be no more powerful than a single neuron, which is incapable of solving linearly inseparable problems.

For spiking neurons, the output function compares the neuron state to some threshold, and if the threshold is exceeded it generates an outgoing spike event.

3.4.3.1 Binary thresholding

In the simplest case the neuron will compare the activity value to a threshold and output a binary value, using either the sign function, or the Heaviside step function:

Sign function:

$$\text{sgn}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

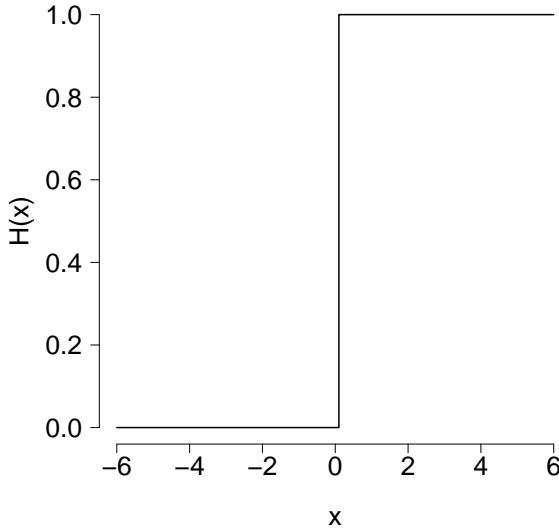


Figure 3.5: A plot of the Heaviside function, showing the threshold at $x = 0$ which results in a discontinuous output transition between $y = 0$ and $y = 1$.

Heaviside step function (figure 3.5):

$$H(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

Thresholding the input to produce a binary “all or nothing” output was common until training techniques that rely on computing the gradient of the activation level were discovered. The sign function (also known as signum, or sgn) is a function that switches from -1 to 1 at $x = 0$ (at precisely $x = 0$ the output value is defined as 0). The Heaviside step function is similar, with the output switching from 0 to 1 at $x = 0$, however, at that point the output is defined as 1.

3.4.3.2 Sigmoid function

The sigmoid function produces an output between 0 and 1 with an “S” shaped curve the gradient of which can be calculated at any point. This function produces a simple behaviour from the neuron; its output will increase in response to increased activity.

Sigmoid function (figure 3.6):

$$s(x) = \frac{1}{1 + e^{-cx}}$$

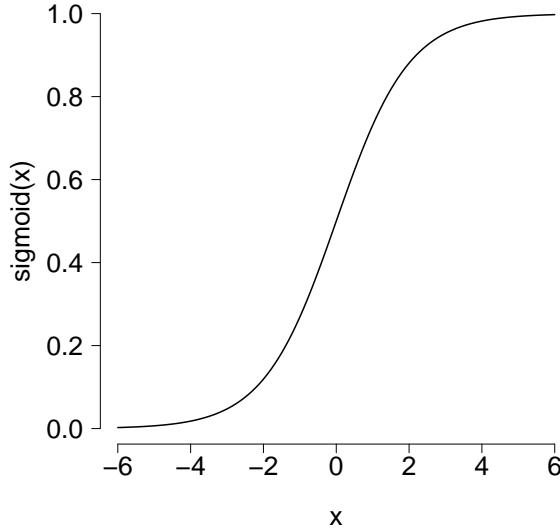


Figure 3.6: A plot of the sigmoid function showing the distinctive “S” shape. The gradient of the curve can be calculated at any point.

where c is a parameter used to control steepness of the curve. In the graph below c is equal to 1.

3.4.3.3 Hyperbolic tangent

The hyperbolic tangent function is often used for neural models which constrain neuron outputs to be between -1 and 1 (though the same effect could be achieved with the threshold or sigmoid functions by translating the output by -0.5 and multiplying by 2). In particular it is used for motor neurons since output forces need a polarity. The parameter c can be used to alter the steepness of the curve.

Hyperbolic tangent:

$$y = \tanh(cx)$$

3.4.4 Spiking models

Spiking neurons (figure 3.8) model the action potential spikes observed in biological neurons. The principal difference between these models and others is that in the spiking models all communication between any two neurons is carried out on a single connection that can only transfer spike events, as opposed to floating-point values, or

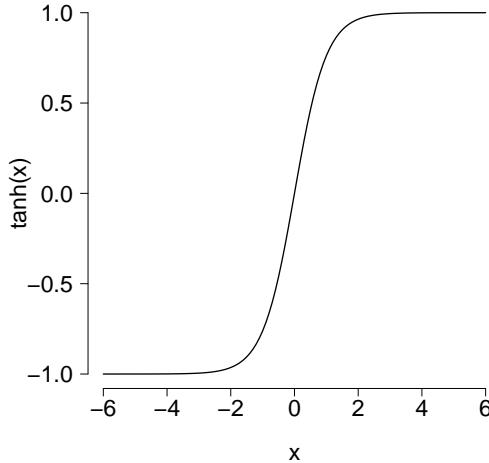


Figure 3.7: A plot of the \tanh function. Like the sigmoid function the plot shows a distinctive “S” shaped curve, and the gradient can be calculated at any point. The dependent variable y is in the range $[-1, 1]$ and symmetric around 0 with reversed polarity.

even binary zeroes and ones. Each spike is a discrete event, and does not have a polarity, level, or any other metadata, meaning that the only information associated with a spike is the time it was generated.

In order to ensure that each spike is discrete, following the firing of an outgoing spike the neuron goes through a refractory period in which its action potential is suppressed, and hence subsequent spikes during this period are either impossible (absolute refraction), or just less likely (relative refraction).

The Hodgkin-Huxley model is widely accepted as a realistic model of action potentials in a biological neuron. It is complex, in that it uses four variables and their respective differential equations. Due to this it is not directly used in synthetic networks; rather, the equations are reduced to simpler single variable approximations which can be more quickly computed on standard hardware. The integrate-and-fire model and the spike response model are the two most common single variable models used for synthetic spiking neural networks [155].

3.4.4.1 Integrate-and-fire model

The leaky integrate-and-fire neuron is excited and inhibited by weighted input connections. If the activation exceeds a threshold it fires an action event. Some proportion of

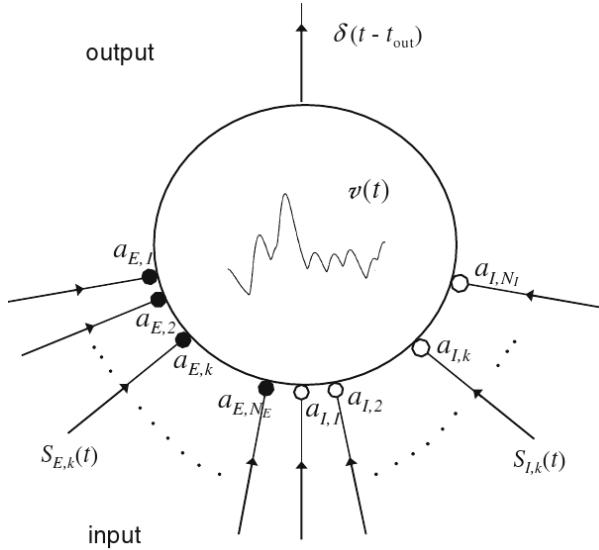


Figure 3.8: *An example spiking neuron. Excitatory connections have black circles, inhibitory have white. Connections are weighted. Incoming spikes contribute to an internal activity value which is thresholded using the Dirac delta function to produce outgoing spikes.*

Credit for image: A. N. Burkitt [52]

action potential is constantly lost to leakage. In the absence of incoming spikes, activity will fall to the resting potential. In this model, spikes only have an effect when they are generated, so the model does not need to store information about spikes that occurred in the past. This model is defined by a differential equation and some associated functions [52, 471].

Input function:

$$x = \sum_{j=1}^n w_j z_j$$

n number of incoming connections

w_j weight of the connection from neuron j

z_j value of the output function of neuron j

Activity function: the activity variable y is set to zero if the neuron is in the refractory period (absolute refraction), or if the activity threshold θ is exceeded:

$$y_t = \begin{cases} 0 & \text{if } t - t_f < t_r \\ 0 & \text{if } y_{t-1} > \theta \end{cases}$$

If this is not the case, the change in activity is determined by the differential equation:

$$\tau \frac{dy}{dt} = -y + x$$

t current time

t_f time at which neuron last fired

τ time constant

t_r refractory period

θ firing threshold

Output function: the output is instantaneously 1 if the neuron activity exceeds the threshold, causing it to fire (this is often represented using a Dirac delta, or unit impulse, function). Following a single value of 1, the activity function will immediately reset y to 0, and refraction will prevent other spikes from being immediately generated.

$$z = \begin{cases} 0 & \text{if } y \leq \theta \\ 1 & \text{otherwise} \end{cases}$$

3.4.4.2 Spike response model

The spike response model (SRM) uses a function to calculate the momentary value of neuron activation, rather than differential equations [155]. To do this, the contribution of each incoming spike in the past is calculated using a function $\varepsilon(s)$. This function will decay over time, representing the fact that the effect of an individual spike diminishes over time. A typical ε function (figure 3.9) would be [135]:

$$\varepsilon(s) = \begin{cases} 0 & \text{if } s < \Delta \\ \exp\left(\frac{-(s-\Delta)}{\tau_m}\right) \left[1 - \exp\left(\frac{-(s-\Delta)}{\tau_s}\right)\right] & \text{if } s \geq \Delta \end{cases}$$

s time since spike

Δ synapse delay (~ 2 ms)

τ_m membrane time constant (~ 4 ms)

τ_s synapse time constant (~ 10 ms)

After firing, a neuron is subject to a refractory period during which its action potential is suppressed in order to inhibit generation of a subsequent spike for some period of time. This is modelled with an η function (figure 3.10), such as:

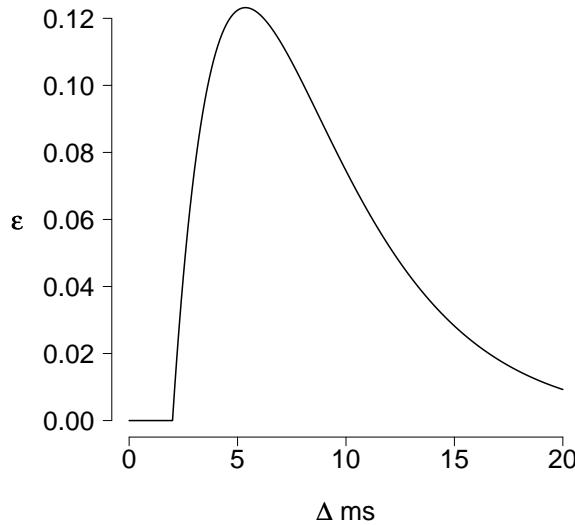


Figure 3.9: *The ϵ function models how an incoming spike, after a period of synaptic delay, has a large effect on the activity of the receiving neuron, but then decays exponentially over time.*

$$\eta(s) = -\exp\left(\frac{-s}{\tau_m}\right)$$

s time since spike fired

τ_m membrane time constant (~ 4 ms)

The ϵ and η functions are combined to give a single function which calculates the activity of a neuron at a point in time:

$$y = \sum_{j=1}^n w_j \sum_{f \in F_j} \epsilon(t_f) + \sum_{f \in F} \eta(t_f)$$

n number of neurons

F spikes of this neuron

F_j spikes of neuron j

w_j weight of connection from neuron j

t_f time since spike f

If the activity y exceeds a threshold θ then the neuron fires a spike, and η is set to -1 to prevent another spike being immediately generated.

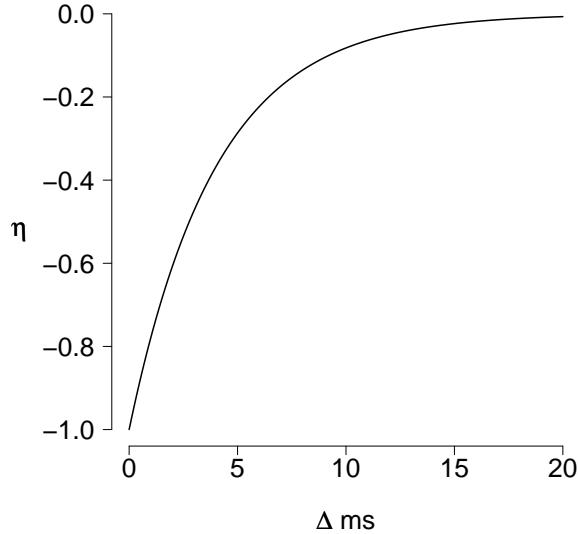


Figure 3.10: *The η function models the refractory period where a neuron's activity is repressed immediately after it fires. The strength of repression decays exponentially.*

3.4.5 Continuous models

3.4.5.1 Sigmoid model

The sigmoid neuron is one of the simplest models, and its dynamics are easy to analyse. The weighted inputs are summed, and then this value is entered into a sigmoid function to compute the output.

Input function:

$$x = \sum_{j=1}^n w_j z_j$$

n number of incoming connections

w_j weight of the connection from neuron j

z_j value of the output function of neuron j

Output function:

$$z = \frac{1}{1 + e^{b-x}}$$

where b is an optional bias value that translates the weighted sum some distance along the activity dimension.

The output of this function relies purely on the current inputs, hence it has no internal dynamics, and the output signal may be discontinuous.

3.4.5.2 Continuous time recurrent model

The “continuous time recurrent neural network” (CTRNN) is a first order derivative model first used for robot control by Beer [18, 19]. It is based on the “leaky integrate-and-fire” continuous model of a spiking neuron [155], the main difference being that in Beer’s model there are no spikes, and hence no threshold, no resetting, and communication between neurons is a continuous activity level instead of discrete events. There is also no refractory period (though that is also optional in the spike integrate-and-fire model).

The activation function produces the gradient of the change in activity between the previous activation of the neuron and the current inputs, biased by an adaption rate (or “time constant”). Integration of the gradient and the previous activity can then be used to calculate the current activity, which is summed with a bias value and placed into the sigmoid output function to calculate the new output value of the neuron.

Input function:

$$x = \sum_{j=1}^n w_j y_j$$

Activation function:

$$\tau \frac{dy}{dt} = -y + x$$

where τ is an adaption rate that alters how quickly the state follows changes in its derivative.

The $-y$ term makes activation “leaky” and tend towards 0. As with bias values, it can be calculated as part of the vector dot product in the input function by considering y as an input with weight -1 , in order to accelerate the operation on vector hardware.

Output function:

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

This is the sigmoid function with b as a bias value.

This model produces continuous dynamical behaviour based on the first order differential equation. Other than that it is functionally equivalent to the sigmoid neuron.

3.4.5.3 Taga’s model

Taga’s model uses two coupled first order differential equations to produce the behaviour of a second order derivative model [307, 427, 428]. The original neural model dates back to a 1914 model of the flexor and extensor muscles of a walking cat, in which a neuron pair provides opposing torques to control each joint [48]. The neuron

pair is modelled as a unit oscillator with the two differential equations. Networks of these paired unit oscillators were used by Taga to model neural rhythm generation in walking bipeds. The differential equations are:

Input function:

$$x = \sum_{j=1}^n w_j z_j$$

Activation functions:

$$\begin{aligned}\tau \frac{du}{dt} &= -u - \beta \max(0, v) + x + b \\ \tau \frac{dv}{dt} &= -v + z\end{aligned}$$

β bias term

τ and $\dot{\tau}$ time constants of inner state and adaptation

b global bias constant

v variable representing degree of adaptation or self inhibition

Output function:

$$z = \max(0, u)$$

The values of τ and $\dot{\tau}$ change the frequency of oscillation, and b changes the amplitude.

This model is parameterised; some constant values obtained from experiments on human biped walking are suggested for its use:

$$\begin{array}{cccc}\tau & \dot{\tau} & \beta & b \\ \hline 1 & 1 & 2.5 & 1\end{array}$$

3.4.5.4 Ekeberg's model

Ekeberg used a third order differential model [122, 350] to simulate the central pattern generators of the lamprey eel. Individual neurons are capable of more complex behaviour than second order models. It is claimed that these third order models are more biologically plausible as they were designed to recreate the dynamics observed in experiments with biological neurons. The model is defined by the differential equations:

Input functions:

$$\begin{aligned}x_e &= \sum_{j \in I_e} w_j z_j \\ x_i &= \sum_{j \in I_i} w_j z_j\end{aligned}$$

Activation functions:

$$\begin{aligned}\tau_D \frac{dy_e}{dt} &= -y_e + x_e \\ \tau_D \frac{dy_i}{dt} &= -y_i + x_i \\ \tau_A \frac{dy_\theta}{dt} &= -y_\theta + z\end{aligned}$$

Output function:

$$z = \max(0, 1 - e^{\Gamma(\Theta - y_e)} - y_i - \mu y_\theta)$$

I_e and I_i the set of excitatory and inhibitory inputs

Θ firing threshold

Γ gain

τ_D dendritic time constant

μ level of adaptation

τ_A adaptation time constant

This model is parameterised; constant values corresponding to four different types of biological neuron, taken from experiment data, are suggested for its use:

Neuron type	Θ	Γ	τ_D (ms)	μ	τ_A (ms)
excitatory	-0.2	1.8	30	0.3	400
contralateral inhibitory	0.5	1	20	0.3	200
lateral inhibitory	8	0.5	50	0	-
motor	0.1	0.3	20	0	-

3.4.5.5 Wave generator models (sine, sawtooth, square)

Wave generator neurons simply generate common waveforms on their outputs, and ignore all signals on their inputs. The waveforms most commonly used are the sine wave, sawtooth wave, and square wave (see figure 3.11). These models are sometimes used as parts of larger neural networks, for example Sims used them along with comparators and other arithmetic logic in his evolved virtual creatures [397, 398]. Hornby used sine waves connected directly to motors in his evolved creatures [200, 204, 205] or neurons with a triangular output function [206] and van Breugel compared sine wave with Bezier curve generators for his evolved ornithopter controller [457]. In Sims's work the outputs of the generators were processed by other elements, forming part of a larger network. In Hornby and van Breugel's works the wave generators essentially were the network — each motor in the morphology took a direct input from a generator

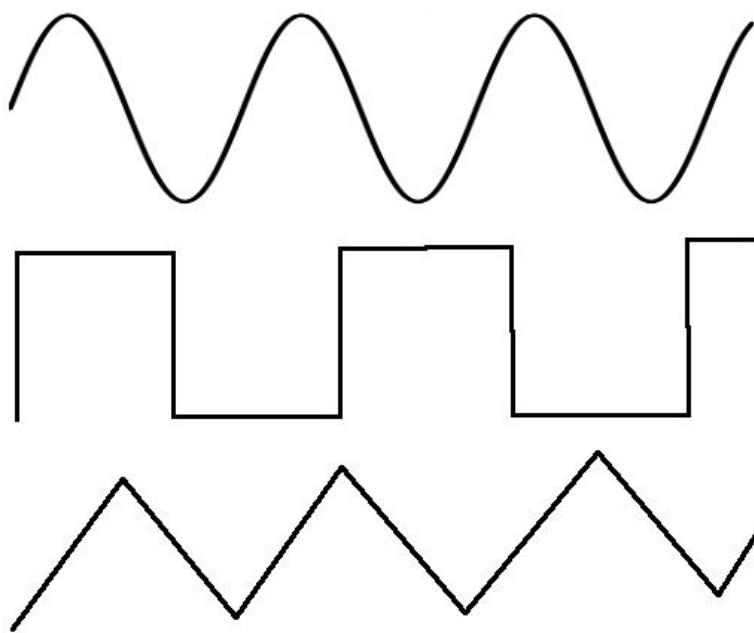


Figure 3.11: *Sine, square and sawtooth waveforms. While the general shape is predetermined, the amplitude and phase can be parameterised.*

node with no intermediate processing. The exception to this was Hornby’s “oscillator neuron” with its default triangle wave output function, but also having the ability to sum input signals and use them to affect the output [206, p.63].

A wave generator model will typically be parameterised with the amplitude and frequency of the wave. For a more complex waveform, such as Bezier curves, the parameters will be the sequence of control points that describe the curve.

3.4.6 Reduced models

The models described above are complex and their implementation requires a large amount of computational resources. There are many ways of implementing such models; either directly through continuous analog circuits, with replicated cells for each neuron, or through circuits in which resources are shared and accessed across a common bus, or through simulation on a digital computer or other programmable logic system.

In all of these cases, if it were possible to reduce the complexity of the neuron model, then the resulting implementation would be simpler and smaller. Several reduced models have been proposed and implemented, although few have been directly

compared to the more complex continuous models. These include boolean networks, multi-value logic networks, and digital logic implementations. Most of these designs have been targeted at hardware implementations [308, 309, 492].

3.4.6.1 Reduced spiking models

Spiking networks transmit discrete events between neurons, and so naturally provide an opportunity to reduce the requirements for computation and communication over continuous models. Models of spiking neurons have been studied for a long time [155, 269], and there have been many implementations in different technologies, in particular VLSI and FPGAs (see section 3.8), where designers have sought to reduce hardware cost by minimising circuit area.

Although the traditional leaky integrate-and-fire model uses single event spikes to communicate between neurons, the computation carried out within synapses, and at the neuron itself, is real-valued. Synapses can be either excitatory or inhibitory, and the strength of a synapse varies. This corresponds to multiplication of the signal by either a positive or negative weight. The post-synaptic potentials are summed, integrated, and the result compared to some threshold. If the threshold is exceeded a spike will be fired along the outgoing axon and the neuron's state (activation potential) will be reset. This is somewhat similar to a continuous network with a thresholded step output function, the only difference being that here a discrete spike is used rather than a continuous maximal value.

In terms of computational requirements, modelling an integrate-and-fire network is as complicated as modelling a continuous time recurrent network; they require the same multipliers, adders, and threshold output logic, so the change from a continuous signal to an event based signal makes little difference. Some attempts have been made to reduce the model complexity in order to reduce the computational requirements for implementation in digital technology.

3.4.6.2 Examples of reduced spiking models

From 1993 to 2001 de Garis attempted to evolve neural networks formed from 3D cellular automata inside custom hardware, eventually settling on machines built from FPGAs (see section 6.9.1). In this model each cell can be in one of two states, which can be interpreted as being the presence or absence of a spike in that time period, effectively making this a spiking model with discrete time. Models with discrete time

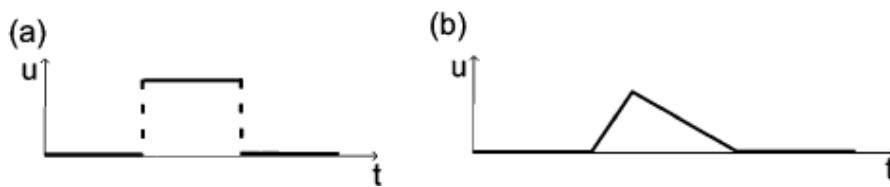


Figure 3.12: *Reduced post-synaptic responses:* a) activity increases by a constant for a fixed time period b) a piecewise linear approximation is used for excitation and decay. Contrast this with the more realistic response of the SRM model in figure 3.9.

Credit for image: Wolfgang Maass [269]

steps have been criticised as not being biologically plausible — they would only be good models of biological networks where, for every neuron, the firing times of those neurons that output to it are closely synchronised [269].

Each cell in a 3D cellular automaton has six neighbours; it will receive an incoming connection from 5 of them, and send its output signal to the other. The cell can either be a neuron, axon, or dendrite. Neurons perform a sum-and-fire function. Axons split a single input signal into five outputs. Dendrites XOR together five inputs and put the result onto a single output. Each connection between cells transfers 1-bit of data.

Neuron inputs are weighted ± 1 [152]. In each neuron a 4-bit accumulator sums the inputs. The neuron activity range was $[-8, +7]$. If the activity fell below -8 the neuron was reset without firing. If the activity exceeded some threshold the neuron would fire and reset to 0. This model was evolved to reproduce some simple digital functions like generation of a pre-defined $00\dots 11\dots 00\dots$ sequence, a 3-bit comparator, and the generation of a 2-bit output that, when used as input to an accumulator, very roughly approximates a single phase of a sine wave (in fact, due to the way the accumulator increments or decrements the counter on each cycle, and the use of only a single cycle for fitness evaluations, the output may have been as simple as a $11\dots 00\dots 11\dots$ sequence, which is a far cry from true approximation of a sine wave) [152]. Later research evolved a SIIC coded sine wave generator, and a SIIC coded phase shift module [316].

In 1996 Maass presented some simplified models which he termed type A and type B [269]. In these models the post-synaptic response of activity increase followed by exponential decay is replaced by piecewise approximations; a piecewise constant function providing a constant increase for a fixed time period (type A), and a piecewise linear function providing a linear increase and decay (type B) (see figure 3.12). Maass showed that type B spiking models are as computationally powerful as continuous models; they can approximate any continuous function with a single hidden layer.

In 2002 Floreano and Zufferey evolved integer spiking neural networks for robot control [133,134,495]. This implementation used an 8-bit PIC microcontroller with an onboard steady-state genetic algorithm and fitness evaluation function. It was used to generate exploration and wall avoidance behaviours, in Floreano's case for a wheeled robot, and in Zufferey's case for a flying blimp, both moving in a rectangular arena. This was a follow on from Floreano's earlier research in 2001, in which he had successfully evolved continuous (non-reduced) SRM neural controllers for a wheeled robot [135] (see page 154).

The neuron model was substantially simplified to be implementable on the limited processor using only 8-bit arithmetic and logic operations. There were eight neurons, and only connectivity and neuron polarity (inhibit or excite) were evolved (actually, to simplify the programming, the network was fully connected but the only allowed weights were 0 and 1, so in effect the topology was evolved rather than some specific weights).

The number of neurons (8) was chosen to maximise performance; the 8-bit microcontroller allowed spikes, neuron signs, and connectivity to each be stored in a single byte, and processed in a single arithmetic operation. The network was updated synchronously, thus doubling the storage required for neuron activity values. Three infra-red sensors were coded onto 8 input signals, which were in turn connected to all of the neurons.

The refractory period, instead of being some non-linear function that inhibits neuron activity, now used absolute refraction in which the neuron's activity was prevented from being updated for some set period of time after firing.

The input function calculated the sum of the inputs, as usual, but here this only required limited-precision signed integer arithmetic; each neuron excited or inhibited (sign -1 or 1), each weight was 0 or 1, and each input value was 0 or 1, to represent a spike being fired in that time-slice.

Once the input function was calculated it was added to the neuron activity value, which again used a limited-precision signed integer operation, with a minimum activity value of 0 being enforced. Exceeding a threshold value of 5 plus some random value in $[-2, 2]$ resulted in the neuron firing and its activity being reset. The addition of a random value was done to prevent the network from converging to a global fixed point attractor state, in which each neuron would be deadlocked waiting for a change in its inputs, and hence the network would cease to generate any useful output signal.

Leakage was simulated by subtracting a constant value of 1 from the activity of

each neuron during each cycle. This constant decrement was much simpler than the exponential decay used in more realistic models.

For details of the signal coding of the infra-red inputs and motor outputs see section 3.3.

In 2003 Xicotencatl implemented spiking neurons on an FPGA by replacing the floating-point multiplier required by each synapse with an integer accumulator which increments or decrements a register by a constant amount for each spike it receives [483]. Whether the value is incremented or decremented depends on the polarity of the sending neuron. If the accumulator reaches some threshold it overflows, and generates an outgoing spike into a post-synaptic comparator, which compares the number of positive and negative post-synaptic spikes, and generates an outgoing spike if it has received more positive pulses than negative. It also acts as a synchronisation element, as the neuron will not fire until the last synapse has fired (although it is unclear whether all synapses must fire, or what happens if the last synapse does not fire).

Xicotencatl showed experimentally that the output-rate of this digital neural network could vary non-linearly depending on the input weight. Although he calls this a transfer function it seems that, since the weights would be fixed, and the distribution of the response to varying frequencies was not characterised, that in fact it acts as a simple stochastic multiplier of the input rate, which would be a linear transfer function. If this is the case, then this design would be limited as a multi-layer network with linear transfer function has no more power than a single layer network. No way of training this network, and no practical applications, were presented.

In 2003 Upogui presented a spiking neuron model optimised for hardware implementation [455]. He noted that hardware implementations of integrate-and-fire and spike response models wasted resources in implementing kernels and numeric integration. This model used integer multiplication, with weights in the range $[-255, 256]$ with 7-bit resolution, so not all values within the range were allowed (e.g. allowed weights would be -4, 0, 4, 8, etc.). Other significant values are the neuron resting potential, threshold potential, and post-synaptic slope.

The neuron behaviour is split into two distinct states — operational or refractory. In the operational state, incoming spikes cause the activity to be increased or decreased by a constant value, otherwise the activity decays linearly. In the refractory state incoming spikes are ignored, and the activity is increased linearly towards the resting potential. Figure 3.13 shows the effect of several input spikes on the activity level, the firing of an outgoing spike resetting the activity, followed by the refractory period.

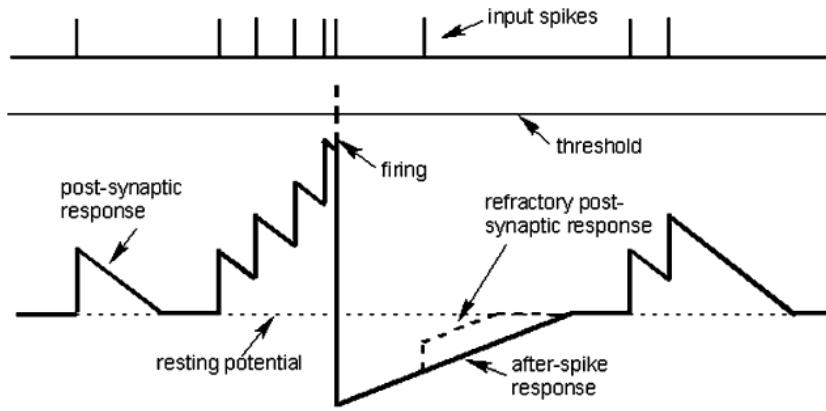


Figure 3.13: *Activity of a reduced digital neuron model. Each spike on a given input causes a constant integer value to be added or subtracted from the activity. The decay and refractory responses are both linear. Networks of this type were evolved to carry out optical pattern recognition.*

Credit for image: Andres Upegui [456]

The model was evaluated on a pattern recognition task, first recognising a 5×5 binary grid, which was presented to a 5-input network one row at a time, with ones being coded as 3 spikes back-to-back, and zeroes being coded as no activity over the same time period. The network was to discriminate between three different input patterns (a cross, X and square), producing a single spike on one of 3 outputs. A similar task involved recognition of the numerical digits from 0 to 9 plotted on a 4×5 grid.

A staged genetic algorithm was used to evolve the integer weights for each connection, and the resting potential, threshold potential, and decay slope gradient for each neuron. Networks were evolved to solve the pattern recognition task perfectly, but they were unable to evolve perfect numerical digit classifiers; the best either had ambiguous firings (more than one output fired), or did not fire at all for some input pattern.

In 2005 Upegui implemented this network model in an FPGA [456]. He implemented a 30 neuron network divided into 3 layers of 10 neurons each, with each neuron being fully connected to the other neurons in its layer, and fully connected to the neurons in the subsequent layer, creating a modular feed-forward topology of internally fully connected layers. Unsupervised Hebbian learning was used to learn a function that discriminates between two input sine waves of different frequencies.

In 2006 Schrauwen implemented a reduced spiking neural network on an FPGA [384] (see section 3.8 for details of the implementation). Lookup tables were used to store a digital approximation of the output function for fast recall. Spikes trigger playback of

constant weight bitstreams, which removes the need for multiplication. In the current implementation changing the weights is difficult (it requires resynthesising the design, or using a larger design that can load weights through a debugging scan chain), so computation is limited to the kind of fixed weight networks evolved by Floreano in 2002 (although they envisage implementing systems based on liquid state machines, which do not require weight training).

Although the model is implemented using bitstreams and serial arithmetic, it is actually very similar to other integer spiking neural networks which are implemented in more traditional ways; the implementation does not change the fact that this is essentially a leaky integrate-and-fire model, although more complex neurons can be constructed by combining various filters. No practical applications were demonstrated.

3.4.6.3 Reduced continuous models

One way of simplifying the model is to reduce the persistent state stored at each node. Some neural models, such as the sigmoid and perceptron, have no persistent node state, so the output of each node is purely a function of its current inputs. However, these models are usually globally synchronous and recurrent, which means that memory is required to store the node output value between clock cycles. The models with node state specify how the state variables change using derivative equations. For the Beer, Taga and Ekeberg models, node state consists of one or more real-valued numbers.

If the state is reduced then the complexity of the activation function can probably also be reduced. Simple binary activation functions can be easily implemented in hardware with a comparator, and discrete step-like functions can be implemented using lookup tables. Many of these approximations attempt to reproduce the sigmoid function. Ultimately the sigmoid curve can be reduced to a binary switch, but more accurate approximations can be obtained by dividing the input into regions, each with a corresponding output from a lookup table, or by using piece-wise linear approximation to transform the sigmoid curve into a series of lines of the form $y = ax + b$, possibly using power-of-two values for a so that the output function can be implemented with shift operations.

Another way to simplify the model is to reduce the complexity of data transfer between communicating nodes. We know from biological neural networks, in which only a single spike event can be communicated at a time between two nodes, that a complex coding can reduce the need for high bandwidth communication. Perceptrons, two state cellular automata, and other neuron models utilising a binary output function

communicate this to receiving nodes as a binary valued signal. Most continuous neuron models use a sigmoid output function that reduces the unrestricted range to a value between 0 and 1. However, within this range the signal is still continuous, and so will require 32 or 64 bits for floating-point values when emulated on a digital system.

The input function computes the weighted sum of incoming values. A digital system simulating a network with real-values requires a floating-point arithmetic unit, which is expensive. There have been attempts to quantise floating-point precision so that these operations can be replaced with integer operations. A further optimisation is to replace integer weights with power-of-two weights, so that multiplication can be implemented with a binary shift operation.

3.4.6.4 Examples of reduced continuous models

In 1988 Chiueh presented a backpropagation based training method for discrete ternary networks [68]. The algorithm requires off-line training of a floating-point version of the network with full floating-point arithmetic. The floating-point network is trained using backpropagation until convergence, then the weights are divided up into discrete thresholded regions to obtain a discrete network. This training method was termed the “multiple-thresholding method”. He also used a genetic algorithm to successfully train these networks, and to perform post-training mixing of (possibly local-optima) solutions.

In 1990 Fiesler presented an extension to Chiueh’s multiple-thresholding method that allowed training to continue using the discrete network, termed the “continuous-discrete learning method” [129]. After following Chiueh’s method a discrete network is obtained. Training patterns are then forward-propagated through this network to calculate the classification error, and the error values are then backpropagated through the floating-point network to obtain new floating-point weights. The cycle then repeats, with the new floating-point weights being transformed into discrete network weights. This process is continued until the network converges, at which point the trained discrete network can be utilised for recall.

Fiesler also presented a modified backpropagation method termed the “direct discretisation method”, in which the network has only discrete weights from the beginning, and the weights are only updated if the delta update value is large enough to move the weight closer to another of the allowed discrete values, thus maintaining a fully discrete network at all times. Fiesler found that on a pattern recall task the continuous-discrete learning method outperformed the multiple-thresholding method,

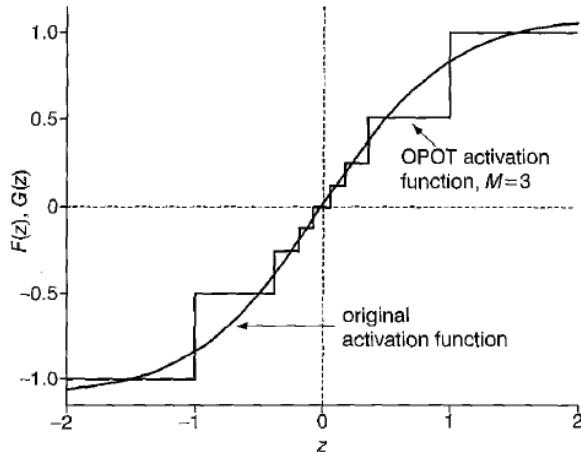


Figure 3.14: A discrete transfer function approximating the continuous sigmoid curve with equidistant regions symmetrical around 0.

Credit for image: H. K. Kwan and C. Z. Tang [251]

and the direct discretisation method failed to work. Increasing the number of discrete states from 2 to 9 reduced the number of training cycles, increased convergence, and improved recall accuracy.

In 1990 Marchesi trained networks with power-of-two integer weights of various precision [278]. This relied on having a continuous version of the network which could be trained using full floating-point arithmetic, and then converting this continuous network into an equivalent discrete network. Marchesi carried out experiments with weight precisions of 0, 1, 4, and 8 (1 to 4 bits respectively), showing that classification error falls as precision increases.

In 1992 Tang and Kwan presented a feed-forward neuron model with power-of-two weights and a discretised sigmoid output function [251]. The output function divided the activation level into non-uniformly sized regions which could be translated into a discrete output in $[-1, 1]$ with only comparison and lookup operations (figure 3.14). Using power-of-two weights meant that multiplication could be implemented using shift operations.

A backpropagation based training method was used, with the error gradient being estimated and constant within each region. The method was similar to that of Chiueh, with a floating-point version of the network being trained, and the weights in each layer being normalised, and then quantised to the nearest power-of-two value. The sum-of-squares error is then calculated, and if it is greater than some threshold, the training loop repeats.

The quantised network model was compared to the floating-point one on an optical number recognition task classifying the integers from 0 to 9. On this task the quantised model was found to perform similarly to the floating-point model, but with a much shorter convergence time when training.

Tang and Kwan presented schematics for this quantised model in 1997, demonstrating that the implementation was both simple and would lead to a reduction in circuit size [432]. In 2002 they showed that their quantised model performed as well as the floating-point one on an optical character recognition task classifying the 26 letters of the English alphabet [252].

In 1992 Kendall described how a quantised network could be trained to perform edge detection in larger images [229]. Kendall's quantised networks used integer weights, thresholds, and boolean inputs and outputs, and hence all operations required for the input function could be carried out by integer arithmetic units. The non-linear output function was the threshold step function, which can be easily implemented with a comparator.

Kendall introduced a training function that optimises a single bit at a time in a manner similar to a “1+1 evolution strategy”. Since this research was using a non-differentiable output function, it was not possible to use a gradient descent algorithm like backpropagation, so instead each bit is flipped in turn, and the performance of the changed network evaluated, and if it performs better the change is kept.

In 1992 Gruau used genetic algorithms to evolve boolean neural networks with binary ± 1 weights and integer thresholds [167]. He managed to evolve networks that reproduced the parity function with 50 inputs, and the symmetry function with 40 inputs. He used a developmental encoding that allows the topology, number of neurons, connections and weights to be evolved together. Gruau claimed the successful evolution of functions with such a large number of inputs demonstrated the superiority of genetic algorithms over backpropagation, as backpropagation does not scale well with the size of the problem. He used the same system to evolve boolean neural networks for robot control in 1994 (see section 6.4)

In 1994 Khan presented a learning procedure for integer weighted and thresholded networks [231]. Khan said that with a binary input network, the first neuron layer could be implemented using an integer multiplier and adder, which suggests that subsequent layers used floating-point arithmetic. The learning procedure added a “distance to closest integer” metric to the error function, and then used backpropagation to train the network. Tests training a few simple benchmark logic functions (e.g. XOR) showed

the integer weighted networks to be as good as continuous for these tasks.

In 1994 Salapura presented parallel and serial implementations of integer neural networks running on FPGAs [373, 374]. In the serial version a single bit is transferred between neurons on each time step. The “delta encoded binary sequence” signal modulation encodes a real-value number in the range $[-1, 1]$ as the proportional of zeroes and ones to output (see section 3.3 figure 3.3). Adding two streams together requires only a 1-bit adder. Multiplication is carried out by a series of 1-bit adders, with the number required being equal to the number of significant bits in the weight value.

In the parallel version the values transferred between neurons were restricted to unsigned 8-bit integers. Weights were 8-bit signed values, resulting in a 16-bit product, which was then added to other inputs to create a 20-bit activation value. The activation was put through an output function to reduce it down to an unsigned 8-bit value. Although training tools were developed, the published papers do not report on any investigations or successful applications of these two network types.

In 1995 Battiti presented a discrete neural network along with training algorithm [16]. The network used continuous weight values from a discrete set, and hence still required floating-point arithmetic. The trained weights were integers with a fixed precision of 2, 4 or 8 bits. The integers were Gray coded, so that a single bit change resulted in only a small movement along one-dimension. These Gray coded integers were converted to floating-point values for use in the actual network.

Battiti’s search algorithm started from a random binary string, and flipped each bit in turn, measured the error of the resulting network, and if it was lower than the current one the search continued from the new network (i.e. it used a 1+1 evolution strategy). A cycle detector prevented the search from exploring areas it had already visited. The training algorithm was compared to backpropagation on the XOR task, and a high energy physics classification task. The training was found to produce networks consisting of only 2 weights, that were substantially better than networks with 4 or 8 weights, and also better than backpropagation on continuous networks. It was also shown that the method could learn a dynamic control task (truck reversal).

In 1995 Ventrella created robots where each joint was controlled by a sine wave oscillator, with the phase offset, amplitude and frequency being evolved by a genetic algorithm [341]. Although not a network, and not possessing any inputs, this could be said to be a very simple reduced form of continuous network, in which the whole network is reduced to a single layer of coordinated sine wave output nodes. Hornby and Pollack did the same for their “genobots” in 2001 [341]. See section 6.14 for more

details.

In 1996 Lundin compared seven different quantisation functions on a variety of benchmarks [267]. He showed that, whilst none were as good as the continuous networks, the performance degradation for many cases was minimal. He tried using various levels of discretisation (2, 3, 5, 7, 15 and 31) and found that increasing the number of levels led to longer training times but lowered the classification error. The symmetrical function performed very well with only binary or ternary states, and both using equidistant levels symmetrical around 0, and scaled power-of-two weights, gave results approaching the performance of the floating-point network with only 15 discrete levels.

Lundin noted that only quantisation functions that result in weight levels that are equidistant and symmetric around zero are suitable for hardware implementation. These weights can be normalised to $[-1, 1]$ by dividing by the maximum weight value, and the normalised values can then be encoded as binary numbers. The scaling down of the weights can be compensated for by scaling up the gain of the activation function. This allows the network to be implemented using integer arithmetic, and if only power-of-two weights are used, then multipliers can be replaced with binary shifters.

In 2000 Plagianakos presented the use of genetic algorithms to evolve the weights and biases of an integer neural network [336]. He experimented with 3, 4, and 5-bit weights, and noted that previous algorithms that relied on backpropagation could not be used with the simpler binary step activation function, and relied on offline pre-training using processors with floating-point units, restricting their use to static problems. This technique used a modified genetic algorithm rather than gradient descent, and hence did not require a differentiable activation function. This was not the first use of a genetic algorithm for discrete networks as Chiueh had successfully used them for training in 1988, and Gruau in 1992, though Plagianakos was perhaps unaware of this work as he did not cite it.

The network was initially evolved offline, and it was found that using sigmoid neurons lowered the evolution time, so a staged evolutionary process was used where the steepness of the sigmoid curve was increased over time until it approximated the step function. The network could then be transferred to an on-chip system, where it could be further evolved to adapt to changes in the problem over time. The evolutionary system was tested, and managed to create some simple networks reproducing the XOR and 3-bit parity problems.

Plagianakos hypothesised that using integer weights helps to prevent over-fitting to the training data. He later compared backpropagation to the evolved integer networks

on some benchmark tests, one of which contained deliberate misclassifications for 5% of the training data [337]. On this single task backpropagation did overfit, whilst the integer networks did not. However, there are techniques for preventing overfitting in backpropagation, by monitoring the classification performance on a test set of data, and by altering the network topology, so a better hypothesis would be to say that an integer network is less capable than a floating-point network with equivalent topology, so in the specific case where the floating-point network would overfit, the integer one may not. A subsequent publication showed that the network topology had been different between the floating-point and integer networks [335].

In 2002 Draghici presented a theoretical result that enabled a lower bound (worst case) for the required precision of integer networks to be calculated [113]. The precision was measured as the absolute integer range required to guarantee the network would be able to converge to a solution as good as a real-valued network; a precision of n corresponds to an integer range of $[-n, n]$.

Experimental results showed that, on all of the three tasks studied, equivalent performance could be obtained with a precision significantly lower than the worst case (figure 3.15). The difference between the theoretical bound and the experimental result could be quite large — in one case, the theoretical worst case precision was 37.2, and the experimentally determined precision was only 5. In the experiments the network topology was arbitrarily constructed using an algorithm which adds neurons to the network as needed. Each neuron used a sign output function with threshold, producing a binary output.

In 2005 Zufferey developed an integer neural network model, known as “PIC-NN”, and used a genetic algorithm to evolve controllers for a flying blimp [493, 494]. The model was implemented on a PIC microcontroller, which, like his 2002 work on reduced spiking models (see page 64), constrained the implementation to a mixture of 8-bit and 16-bit data types and arithmetic operations. Neuron activation values were 8-bit $[-127, +127]$ and weights 4-bit $[-7, +7]$ (the weight is effectively one tenth of this so that no single connection can saturate a neuron). Arithmetic operations were carried out using 16-bit signed arithmetic. The neuron activation function was tanh, not the spiking model used in 2002. Rather than calculate it, values were precomputed for all 256 activation values and stored in a lookup table. Exploration and wall avoidance behaviours were successfully evolved with this model.

In 2006 Plagianakos compared genetic algorithm based learning for integer networks to other learning methods that rely on either backpropagation with estimates

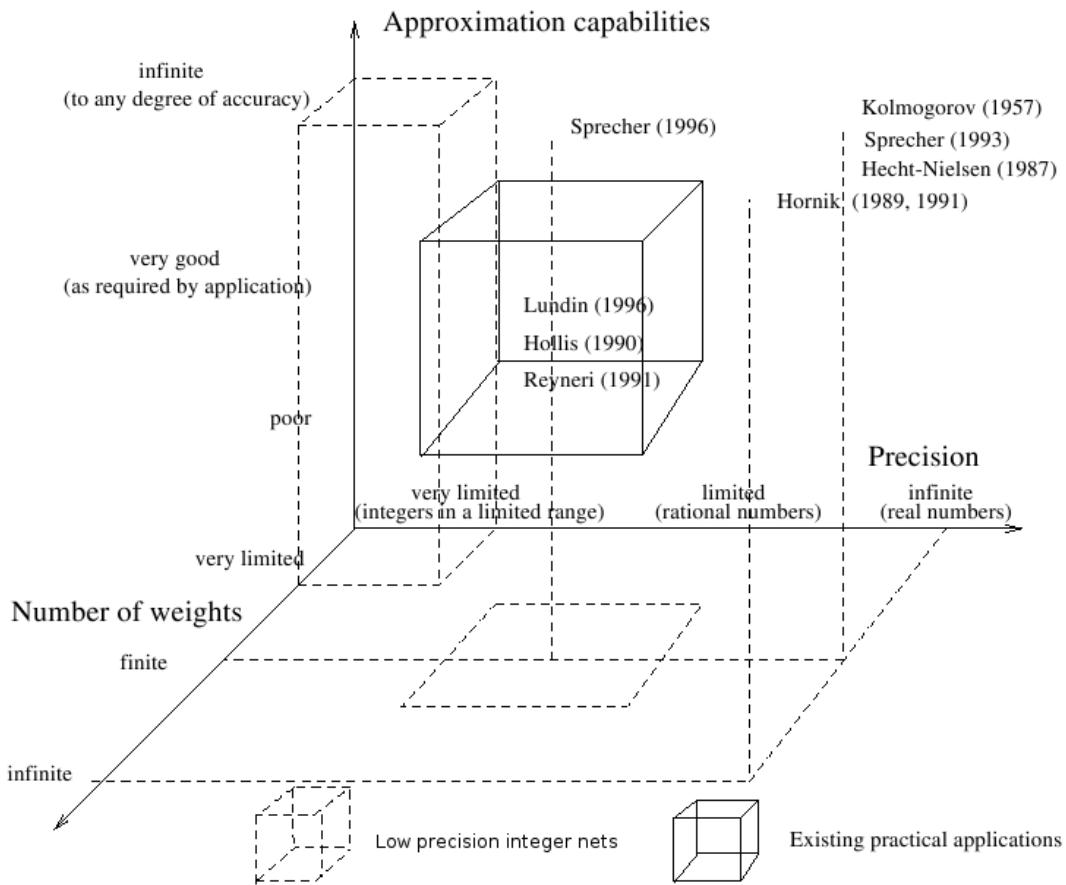


Figure 3.15: *Theoretical results have shown that neural networks with an infinite number of weights of infinite precision can calculate any function. However, traditional implementations (solid bounded box) use a finite number of weights with rational precision. The capability of integers with limited precision (dashed box) has been shown to be as good as those with rational number precision on several tasks.*

Credit for image: Sorin Draghici [113]

of the error gradient obtained from a continuous version of the network, or training a continuous version of the network, and slowly steepening the sigmoid curve until it approximated a discrete transition [335]. He used the same benchmarks as in 2002 (XOR, parity, and MONK), and introduced a new dynamic control benchmark for a controller that specifies the force to be applied by an industrial cutting machine. On all benchmarks the genetic algorithm learnt the classification function better than the other algorithms, although sometimes requiring significantly higher computational effort.

3.5 Computational power

Neural networks are computationally universal. Although it was famously shown by Minsky that single perceptrons can not calculate the exclusive OR function [301], a two layer perceptron network can implement any logical function. The same is true of more complex neural models with sigmoid output functions. From networks implementing digital logical functions a universal Turing machine can be constructed [395].

It has been shown that any bounded continuous function can be approximated by a sigmoid feed-forward network with only two hidden layers, and any arbitrary function can be approximated with only three hidden layers [80]. The accuracy of the approximation depends on the number of nodes in each layer, the connectivity and the connection weights. Continuous time recurrent networks with first order neuron activity functions have been shown to be of equivalent power to any higher order model [396]. It has been shown that an individual spiking neuron is more powerful than a sigmoid neuron, and that any function that can be computed by a sigmoid network can also be computed by a spiking network [269].

The equivalence of the simple sigmoid network model with arbitrary functions, higher order network models, and spiking models means that, for the purpose of creating synthetic neural networks, no increase in computational power is gained by the use of higher order or spiking models. With respect to the use of genetic algorithms to evolve dynamic robotic motion, the increased success reported by some researchers in using higher order neuron models [349], or in using spiking models [135], does not derive from a greater computational ability inherent in these models, but instead must derive from an increased probability of forming neural structures that act as central pattern generators producing dynamic cyclic waveforms. Thus, it would not be necessary to choose one model over another if the genotype encoding, representation, and evolutionary algorithm structured the search in a way which enabled analogous

network structures to be discovered regardless of the specific neuron model in use. This would likely involve a high degree of modularisation of the evolving network genotype, and the ability of the genetic search to exploit discovered structures with parameterised variables, essentially allowing a search with sigmoid neurons to construct complex parameterised blocks from several interconnected neurons, and to easily reuse these blocks. If the more complex neuron structures are indeed fundamental building blocks, then an appropriate genetic algorithm would re-create them from scratch.

3.6 Robot control

All of the continuous neuron models presented here have been used to successfully control locomotion in three-dimensional articulated robots; for Beer's continuous time recurrent model see [18, 355], for Taga's 2nd order model [307, 427] and for Ekeberg's 3rd order model [41, 349, 350].

Despite the widespread use of neural network controllers in robotics such a universal computational framework is not strictly necessary, and for many control tasks pattern generators with no inputs, and with outputs connected directly to motors, have been shown to be sufficient. Examples include generators of sine waves [200, 204, 205] and parameterised Bezier curves [457].

Low-level motor control, carried out by neural networks, can be supplemented with higher level behavioural control [344]. In evolution research this is usually done by using a staged genetic algorithm, in which the lower level network is frozen, with secondary networks being allowed to only modulate, but not alter, the pattern generation networks connected to the motors [238, 349]. Traditional robot control architectures would implement action selection mechanisms to arbitrate between multiple, competing, mutually-exclusive behaviours. Non-traditional control architectures, such as those from the field of behaviour-based robotics, have also been used to successfully control walking robots [47, 181]. It has been shown that action selection can be carried out in dynamical systems, like neural networks, as a result of non-linear phase transitions [415].

3.7 Training

For the neural network to perform some useful function the values of various parameters have to be set. Spiking neurons have activity thresholds. More complex models

have coefficients that determine the rate of state adaptation, excitation and leakage.

Each connection usually has an adjustable weight, though some systems use fixed weights and modify other parameters, such as connectivity, instead. In traditional neural networks the sigmoid neuron is used with no parameters, so network behaviour is completely determined by the connection weights. Data sets of corresponding input and output values are used to “train” the network by adjusting the weights until the network produces the same output on a given set of inputs as the training data.

The most successful systems employ progressive evaluations and adjustments to the network, eventually stopping when the network performs above some threshold of accuracy on the given task. The main problem with network training is credit assignment — when an incorrect aggregate output is observed from the whole network, how do we determine the contribution of, and adjust, individual neurons to correct it, without over-generalising or incorrectly changing the classification of other inputs? The first, and most successful, system to solve the neuron credit assignment problem was “backpropagation”, which was introduced in 1986.

3.7.1 Backpropagation

Backpropagation [368] has been the most successful training method for generic classification and pattern matching tasks. It requires a training set of input patterns and corresponding outputs. The input patterns are presented to the network and the output observed. If the output is incorrect the weights are tuned until the correct output pattern is observed. This process is repeated for every input and output data pair in the training set.

When an incorrect output is observed each neuron’s contribution to the output is estimated. This depends on three things: the value the neuron is currently outputting, the depth of the neuron in the network, and the weights on its outgoing connections to other neurons. If a neuron’s output is 0 then it is neutral and will not be affecting the incorrect network output. If the neuron’s output is above 0 then it will be effecting the output, with the effect becoming greater as the output value rises to 1, then it will be strongly affecting the overall network output. The closer the neuron is to the final output node, and the stronger its connections to subsequent neurons, then the higher its contribution to the final classification will be.

The gradient of the neuron’s activation function is then used to calculate how much the outgoing connection weight should be adjusted by, weighted by the neuron’s esti-

mated significance and a global learning rate. Sigmoid neurons are used as they have an easily computable gradient. This weight adjustment is performed for every connection of every neuron, moving the network output closer to the correct output. Training then continues on the other input output data pairs.

The whole process is repeated until the weights converge. At this point the network may or may not correctly classify all of the training data — it may be that it is not possible for the network to classify all the network data, or the network may have become over-trained, classifying the training set perfectly, but not generalising well on other data. The network's performance should be evaluated on a different (non-training) dataset.

There are a few problems with backpropagation that motivate the search for better algorithms. It is not clear how to create the network topology. Choices such as the number of neurons, and the degree of interconnectedness, are made by intuition, trial and error. Backpropagation is difficult on networks which contain loops and feedback. How can the significance of a neuron's contribution to the network's output be estimated when it can excite and inhibit its predecessor neurons, which in turn stimulate it, in a continuous loop?

Transforming the cyclic network into a feed-forward one by unrolling cycles, training it, and then back-annotating the weights to the cyclic network is one possible approach, but there are problems with state explosion — how many times should a neuron within a cycle be replicated before its contribution to the network dynamics is accurately accounted for? There is a more fundamental problem that the dynamics of recurrent networks are capable of forming not just point attractors but also oscillatory behaviour caused by limit cycles, and there is no way for a feed-forward network to model this.

Feedback is an essential element of control theory, and is likely to play a large part in any neural network of significant worth. Fully interconnected networks, where every neuron is connected to every other neuron, are the most expressive, since every other topology is a subset of full connection, yet backpropagation is unsuitable for training these networks. Another problem with backpropagation is that it limits the activity and output functions by requiring them to be differentiable.

3.8 Hardware acceleration

Synthetic neural networks, or a computational framework to support them, can be implemented directly in silicon. Neural network hardware has been implemented in either full custom analog VLSI [381], mixed-mode digital and analog [379], digital “application specific integrated circuit” (ASIC) [434], a combination of digital CPU with programmable analog array [128], a “field programmable gate array” (FPGA) [185, 384, 447], or reprogrammable transistor logic such as a “field programmable transistor array” (FPTA) [417, 449] or “field programmable analog array” (FPAA) [24, 212]. Lately, the programmable “graphic processing unit” (GPU) pipelines of 3D graphics cards have also been utilised [27, 353]. In mobile robotics the neural network is often simulated using an embedded microcontroller to reduce power consumption (see section 6.4).

Due to the wide amount of research done on this topic over the last few decades, only a few notable designs will be presented here. They may be notable because they are recent, historically important, or were combined with some other topic of this thesis, such as genetic algorithms, or all-digital non-floating-point implementations (typically, using spiking neurons).

The implementation of neural networks designed at the analog circuit level is known as “neuromorphic engineering”. This kind of design is expensive and time consuming, as circuits must be designed and routed at the level of analog transistors, and simulated using slow ”simulation program with integrated circuits emphasis” (SPICE) [334, 346] based simulators. This was a popular research topic in the 80s and early 90s, but recent research has focussed more on programmable logic and generic CPUs, as designs based on these technologies are much cheaper to create and manufacture. The most famous examples of this design style are those produced by the Carver Mead group at Caltech, who have implemented vision processing systems and neural models from hand-designed components that model the neural structures found in biological brains [287].

In 2002 Schäfer presented a platform for the simulation of large networks of spiking neurons [378]. The system was built primarily from DRAM, FPGAs, and ALUs, although a later parallel implementation replaced the main processor with a DSP. A single neural processing board, interfaced to a workstation, could simulate 130,000 neurons with 16 million synapses, and 8 of these boards could be combined to simulate over 1 million neurons. The simulation was event-based, aiming to minimise unnec-

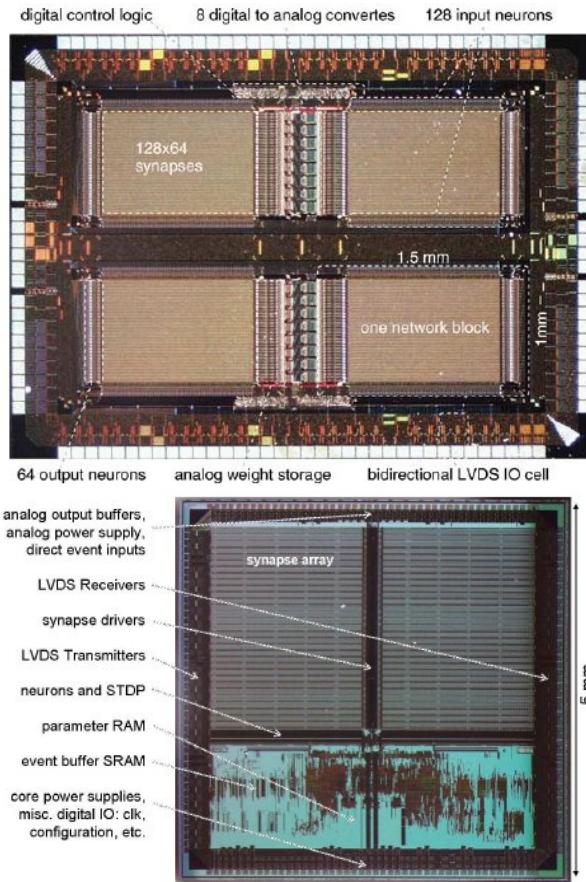


Figure 3.16: *Top: The HAGEN neural network ASIC*
Bottom: A spiking neuron ASIC with synaptic plasticity
Credit for image: Johannes Schemmel, Kirchhoff-Institut für Physik [379, 380]

essary computation. The DSP based implementation supported 64 processor boards, with each neuron being updated every 5ms.

The “Heidelberg analog evolvable neural” (HAGEN) ASIC (figure 3.16) is a mixed-mode analog neural network architecture developed between 2004 and 2006 [379,380]. It consists of analog circuits that emulate neural network dynamics, connected to a traditional von Neumann architecture with a combined FPGA and PowerPC CPU. It is capable of on-chip evolution of neural connection weights, allowing the system to compensate for voltage, temperature, and manufacturing variations at runtime [381]. The designers note that analog VLSI presents a difficult environment for the implementation of neural networks, as traditional training algorithms such as backpropagation rely on knowledge of the first derivative of the neuron transfer function, which will vary on an analog ASIC due to manufacturing tolerances, voltage deviations, and temperature.

In 2005 Schürmann announced a mixed mode neural network ASIC based on liquid

state machine concepts [385]. It had 256 McCulloch-Pitts neurons with 33,000 analog synapses and programmable connectivity. He confirmed that optimal computational activity occurs at the “edge of chaos” (see section 4.8).

In 2006 Schrauwen presented details of a spiking neural network implemented on an FPGA [384] (also see section 3.4.6.1). This was a wholly digital design that used pre-computed lookup tables (LUTs) to calculate the output function, rather than real-time floating-point arithmetic, and a serial arithmetic unit for other operations, rather than a larger parallel unit. The use of spikes reduced the requirements of inter-neuron signalling, and provided an opportunity to use serial adders, since only single-bit events are transferred at a time.

Each connection has a constant weight associated with it, stored in a serial shift register. An incoming spike triggers replay of the weight as a serial bitstream. These serial bitstreams are added using serial bit adders to calculate the input function. Integration is then carried out using a shift register, with the weight generated by each spike being added to the currently stored value, and the result fed back into the shift register.

The architecture is pipelined to increase throughput. When implemented in a commercial off-the-shelf FPGA it can simulate 1400 neurons almost 6000 times faster than their biological counterparts, meaning that it would be a cheap and practical platform for real-time neural modelling. The design has not been used in any practical applications, presumably due to its lack of adaptive connection weights.

Recent advances in graphics card technology have led to the development of the “graphics processing unit” (GPU). This is a processing unit that is custom designed for pipelined high speed parallel floating-point operations. If a computational algorithm can be translated into a set of video operations, such as the application of a convolution function to a framebuffer image, then it can be implemented on a GPU.

The development of high speed GPU libraries is now a research field in its own right. There have been several successful attempts to utilise GPUs for neural network simulation [27, 352, 353], each reporting an approximate 10 to 20 times speedup over conventional CPUs. One of these included a cluster based implementation which significantly outperformed a CPU-only implementation [27, 353].

3.9 Summary

This section has discussed synthetic neural networks. Synthetic neural networks are mathematical models of the real world networks observed in the brains and nervous systems of biological creatures. There are two primary motivations for researching and creating them — firstly, to model real biological networks, and secondly, to create computational systems that can classify and process noisy input data and hence perform useful tasks like pattern recognition.

A variety of mathematical models have been created by previous researchers, modelling individual neurons and their connections at different levels of abstraction. Most of these models are claimed to have a biological basis. The abstract mathematical models utilise continuous valued variables, but are simulated on digital computers using floating-point arithmetic, which is inherently discrete. Biological networks and mathematical models of biological networks are temporally continuous, whereas networks are simulated on digital computers in discrete time steps. More complex models, with more variables and smaller time steps, can simulate biological networks with greater accuracy, but this simulation requires greater computational resources. The question arises as to whether simpler networks with smaller computational resource requirements are able to carry out the same functions as the more complex models, and if so, how degraded their performance will be. Some simpler versions of common neuron models have already been developed, using limited precision arithmetic operations in place of floating point arithmetic, but there have been few attempts to test their performance on real world problems.

This thesis explores whether simpler quantised neural models can be used in place of more complex ones for some standard robot control tasks, and attempts to quantify the performance difference between these models. This chapter has provided an overview of complex and reduced neuron models that have been used in previous robotics research.

Chapter 4

Other networks

In the previous two chapters we have covered both biological and synthetic neural networks. This chapter will describe other types of network which also perform various types of distributed parallel computation.

The defining characteristic of a neural network, as opposed to these other networks, is that each neuron has a state defined by some number of continuous equations, and an output value defined by the output equation. Having said that, even this primary characteristic does not completely differentiate neural networks from the various other network types, as analog circuits and continuous cellular automata could also be said to have node states based upon continuous values and update equations.

The various parameters that differentiate these networks from each other are:

topology Although in theory the nodes of a network can be connected in any topology, a defining characteristic of some of these networks is a specific type of topology. Even though this is the case, researchers often mix topologies in order to conduct new experiments. For example, cellular automata as originally defined have a fixed topology, with all nodes having identical connectivity. However, non-uniform cellular automata were later devised, in which each node has a unique connectivity.

node state The node state can be represented by one or more variables, which are either floating-point or discrete. Integer based states are a subset of discrete, where the number of states is equal to the number of possible integers. Again there is crossover in this area, for example cellular automata researchers have used both continuous and discrete state systems.

node update function Closely related to the node state, is the node update function.

It can be uniform for all nodes (as in classical cellular automata), or can vary between nodes. It can be based on one or more ordinary differential equations simulated using integration, as in neural networks, or can be a logical function, as in digital circuits and cellular automata.

update order This varies from truly parallel (e.g. with specialised circuitry such as analog circuits), to synchronous (such as traditional cellular automata and boolean networks), or asynchronous (also used in cellular automata and boolean networks). With synchronous updating the state of every node is (or appears to be) updated simultaneously. New output values derived from the new state are not visible to other nodes until the next synchronous update. With asynchronous updating nodes are updated in sequence, with the node updated in any given time step being randomly chosen according to some probability distribution. In a parallel system data values are read, and new states updated, without reference to any timing information; this is in contrast to asynchronous systems, which may use control signals between individual nodes to coordinate data transfer. In truly parallel networks, like analog circuits, data is often temporally and spatially continuous.

signal delay The timing model affects when changes in a node's output become visible to other nodes. Time delays in connections between nodes are particularly important for accurate simulation of biological networks and electronic circuits, as these are models of real systems with physically constrained propagation delays, such as dendritic and synaptic delay, or wire capacitance and length.

4.1 Analog circuits

Analog circuits are created by connecting various components, including transistors, resistors and capacitors, with conducting wires. Electrical current is then applied to the component network, producing an activity pattern modulated by the components. Designing analog circuits by hand is difficult and time consuming, hence designers use a process of abstraction to create libraries of modular components. One example of this is the construction of the pre-laid out logic gates, such as XOR, NAND, etc. used in “application specific integrated circuit” (ASIC) design.

Analog electronic circuits have some similarities to neural networks. They generally consist of a large number of nodes arranged into a modular hierarchy. The state

of analog circuits is precisely defined by the voltage present on all of the wires and capacitors. In ASIC design the wires are used to join transistors together. Each transistor acts as a non-linear amplifier. This creates a large scale network consisting of small, simple units. The basic transfer curve of a transistor as it switches is also similar to the transfer function of a biological neuron.

As analog circuits have a complex, non-linear dynamics, they are difficult to design and analyse. Despite the complexities of designing dedicated analog circuits, they have been successfully used for robot control, including complex tasks like the control of three-dimensional flight in miniature robots [138]. Tilden's BEAM robotics methodology proposes that robust and complex robot behaviours can be built from networks of analog components [181], although the use of TTL logic does make the networks somewhat digital. It has been successfully used to construct walking robots [416].

4.2 Digital circuits

Digital design consists of creating electronic circuits from components that represent the basic boolean operators AND, OR, NOT, XOR, and state holding elements such as flip-flops and latches. Digital circuits have been used to successfully control walking robot [47].

Human designers form an abstraction layer by creating components which strictly obey external models of behaviour, and whose interactions with other components are similarly strictly defined. For example, in a digital system the continuous voltage domain is turned into an abstraction consisting of only two states, corresponding to the boolean values of false and true (or 0 and 1). This is done by setting a threshold value which splits the continuous voltage domain $[0, 1]$ into two regions. Around the threshold there may be an unstable area where the output is undefined.

The difficulty of dealing with the continuity of time is dealt with by dividing time into discrete periods. A central clock generator provides a synchronised timing signal to each state holding unit of the system. Upon receiving a particular timing event (traditionally, a rising edge of the clock signal) these units will update their state simultaneously .

Asynchronous circuits are a form of digital circuit that do not require time to be broken up into discrete steps. Instead they introduce new primitives that are used to manage the continuity of time and break it down into an event based model. It is generally accepted that asynchronous circuits are harder to design and analyse than

synchronous ones due to the lack of a globally unique state.

4.3 Asynchronous circuits

Currently, almost all digital circuit designs are synchronous, meaning that they divide time into discrete periods, and utilise a single, global clock signal to mark the division between these periods, allowing state holding components to synchronise their data transfer. Asynchronous design [84] is the art of creating circuits which operate without a clock signal. There are many ways in which this can be done, including generating local clocks for sub-circuits, such as in the “globally asynchronous locally synchronous” approach, or by using hazard-free logic. These approaches decentralise control and eliminate the global clock, whilst preventing logical hazards and maintaining correctness.

Asynchronous design encompasses many different models of delay, from “delay insensitive” where the circuit behaves the same irrespective of wire and gate delays, through to “bundled data”, where the latency of combinational circuits in the data path is pre-calculated and gates inserted into the control path to delay signals by some time greater than the worst case.

Delay insensitive circuits are attractive because of their consistency; they exhibit the same behaviour over all technology processes, and over a wide range of operating voltage and temperatures. Bundled data circuits tend to be smaller, but require precise control over routing and transistor placement.

Asynchronous circuits are closely related to asynchronous cellular automata and asynchronous boolean networks. They do not have the uniform topology of cellular automata (although this is countered by non-uniform cellular automata). The main difference is that asynchronous circuits possess two new primitives that can not be represented with a simple combinational function: the C-element and the arbiter, which are both used to manage time by combining transitions on their inputs into a single synchronised output event. These elements could be constructed from multiple cells in a boolean network, or in a non-uniform cellular automata with appropriate rules, but, as with the digital C-element in figure 4.1, these implementations may require careful consideration of timing. A method for translating asynchronous circuits to two-dimensional asynchronous cellular automata was presented in 2003 [331].

A C-element (figure 4.1) is functionally similar to an AND gate. When the inputs are 11, the output is 1. When the inputs are 00, the output is 0. The difference lies in

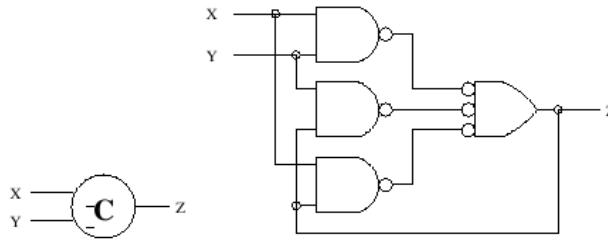


Figure 4.1: *The C-element is used to control signal timing in asynchronous circuits. It has a schematic glyph (left), and can be implemented from digital gates inside an ASIC or FPGA (right).*

Credit for image: Al Davis [84]

the intermediate output, when the inputs are 01 or 10. An AND gate would output 0. A C-element, however, retains the state of its last state when both inputs were equal. For example, if the inputs were 11 (causing output of 1) followed by 10, the output would still be 1. If the inputs are 00 (causing output of 0) followed by 10, the output would still be 0. Hence the C-element stores a state, and state transitions occur upon a 00 or 11 input pattern.

An arbiter has two inputs and two outputs (figure 4.2). Active (value 1) outputs are mutually exclusive, or alternatively both outputs can be inactive (value 0), so the allowed patterns are 00, 01, and 10. Each input corresponds to one output, when the inputs are 00 and one switches to 1 its corresponding output will become 1. If an input rises and the other is already active then the rise of its corresponding output will be delayed until the already active output falls. If both inputs simultaneously switch to 1, there will be a period of resolution, followed by the system settling in a state where only one of the outputs is active. During the resolution period the outputs are stable; a 11 pattern will never occur. This useful feature allows arbiters to be used to control exclusive access to shared resources by serialising access requests.

It is often claimed that asynchronous designs provide several advantages over synchronous ones [84]. These include improvements in average case performance, lower energy consumption, improved modularity, scalability and re-usability, and lower electromagnetic emissions leading to increased resistance to security attacks based on side-channel information leakage.

Asynchronous designs that use completion detection circuits, or that use sub-circuits which can generate completion signals, will exhibit average case behaviour. Synchronous circuits always exhibit worst case behaviour since the clock period must be longer than the time taken for signals to propagate along the critical (slowest) path be-

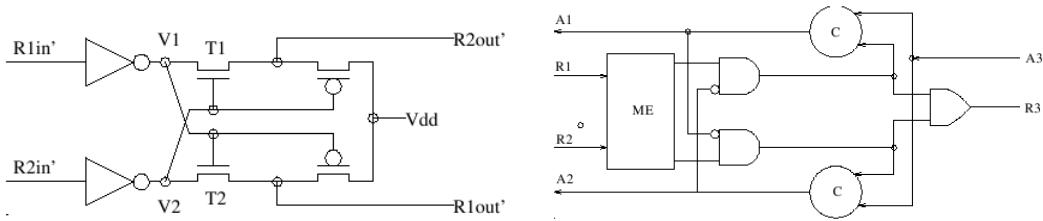


Figure 4.2: *Left:* A mutual exclusion (ME) circuit only allows one output to be active (value 1) at a time. *Right:* An arbiter is used to serialise competing requests for access to a shared resource; it can be constructed from a mutual-exclusion (ME) circuit and C-elements.

Credit for image: Al Davis [84]

tween any two state holding elements. Practical experience has shown that it is difficult to exploit this advantage for two reasons — completion detection modules for delay insensitive circuits increase circuit size and signal delay, and in bundled data circuits every path must use the worst case propagation delay anyway.

Asynchronous circuits can be more energy efficient than synchronous ones. Modern designs use CMOS transistors which draw current when switching but only draw a very small leakage current when in a stable state. In asynchronous circuits, changes in input signals propagate, causing power to be drawn, but when the inputs are stable the circuit is stable, so there is no switching and little power is consumed. In contrast, synchronous designs allow logical hazards and races to temporarily propagate through combinational circuits. The clock state ensures these invalid values will be ignored by state holding components, but their propagation results in unnecessary transistor switching, and hence wasted power.

Pipelined designs offer further opportunity for power saving. Synchronous pipelines propagate “bubbles” when no new data is available, causing each stage to consume power. Asynchronous pipelines have no upper bound on the time taken between data entering and leaving each stage, their timing is “elastic”, so there is no need for pipeline bubbles.

All logic and wire paths exhibit slight variations in their delay due to process variation, operating temperature, and power consumption. As these circuits are combined to create larger circuits the paths through them become longer, and hence the problem of additive skew increases as the delay variations are combined. The flow control of asynchronous circuits eliminates this problem.

Asynchronous circuits can be connected together regardless of fabrication process

or any other aspect of the implementation, making it easy to reuse existing designs. In contrast, synchronous designs must be verified for different processes and clock speeds. True delay insensitive circuits can even have their layout and wire routing changed without affecting functionality. Lower electromagnetic interference makes it easier to mix asynchronous circuits with analog and radio circuits [145].

Routing a global clock signal to every latch in a circuit is a barrier to scaling up to very large designs. This problem is exacerbated as feature sizes get smaller and wire delays begin to dominate over transistor delays. It is unlikely that a global clock will be feasible in deep sub-micron designs such as super-conducting “rapid single flux quantum” (RSFQ) technology, which enables flip-flop transition speeds of 770 GHz [62]. At this frequency it could take hundreds of cycles for a signal to propagate the length of the chip, which severely limits the area reachable within a single clock cycle. More importantly, at such a high frequency, delay variations due to process and operating temperature can be greater than the clock period. Asynchronous circuits have no global signal to distribute and have been successfully used for high speed RSFQ designs [103, 326].

Security can be increased by using asynchronous circuits. Synchronous circuits can exhibit power fluctuations dependent on the clock and data. Side channel attacks have been developed which monitor these fluctuations and use them to recover key data from cryptographic applications. This is a particular problem for smart cards which are used to store digital cash or control access to pay-TV services. Asynchronous circuits distribute processing across time more uniformly, reducing electromagnetic emissions and making timing based attacks more difficult.

4.4 Genetic regulatory networks

A genetic regulatory network is a formalisation that views the activity of a group of genes that can affect each others transcription as a network (or directed graph). Each node in the network corresponds to a single gene. The activity of a gene can indirectly turn the transcription of other genes on and off if its corresponding protein binds to the upstream regulatory region of a gene. Where a gene influences another in this way, there will be a directed edge between them in the network. If a gene does not influence another, then there will be no edge between them. If two genes must be active to influence a third gene, then this will be represented by the presence of two edges in the network, one from each independent node to the dependent node. In this way, the ac-

tivity of a group of genes in a living cell can be viewed as a network (or directed graph). Genetic regulatory networks are modelled using several formal structures that differ in the amount of state they allocate to each node (continuous, multi-value or boolean), and in the function that is carried out at each node to determine changes in state and output values. These differences affect simulation speed and accuracy — more accurate simulation models will generally require greater computational resources. It has been argued that some of the simpler and faster models enable dynamical behaviour that is biologically impossible [179, 371].

The development of a biological life form relies on the ability of its cells to metabolise large molecules, and from what is left synthesise the molecules it requires in order to sustain its life and reproduce. Each non-developmental cell has a specialised “cell type” which it arrives at through a process of cellular differentiation. These cell types perform various functions, such as the communication and computation of a neural cell, or the physical contraction of a muscle cell. The life of a cell, from creation to death, is guided by genes contained in its DNA.

Each gene is converted into a unique protein by the machinery of the cell. A protein can catalyse a chemical reaction that sustains the cell’s metabolic pathways (i.e. breaking down food or constructing new proteins and acids), can perform cell maintenance functions (e.g. maintaining cell structure, signalling, molecule transport) or can bind to the DNA and either activate or inhibit the production of other proteins. Proteins that affect the transcription of a gene by binding to its upstream regulatory region are known as “transcription factors”. The function of regulating the transcription of genes by binding to DNA is analogous to the state changes that an electronic controller will perform. DNA can be viewed as a state machine, with proteins being its output signals. In a digital controller, a multi-valued output signal can be in one of several states, similarly the concentration levels of a protein within a cell can also be grouped into different levels. During state changes an electronic controller will produce outputs which determine its next state. In a cell, genes will be converted into proteins, some of which will in turn bind to the DNA, thus determining the next state. In an electronic controller the signals which do not affect the state of the controller will instead perform functions regulating the datapath, or communicating with external devices. In a cell, proteins which do not regulate DNA will instead regulate the metabolic pathways, carry out mechanical functions or communicate with other cells.

Cells and reactions are affected by concentrations of molecules. Although the concentrations of molecules are discrete (we never have a fraction of a molecule), they

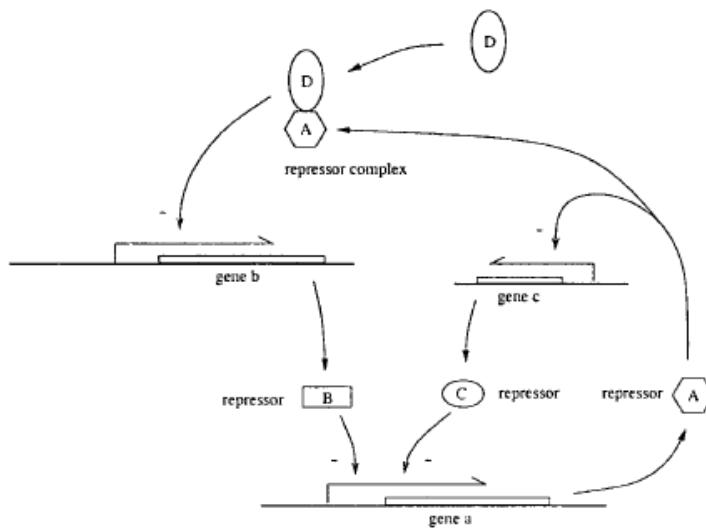


Figure 4.3: An example genetic regulatory network containing three genes which are transcribed to produce proteins (A, B, C, D) that act as transcription factors by binding to the upstream promoter region of specific genes, which suppresses transcription of those genes. The horizontal arrows show the direction of transcription. A and D are both required (AND relation) to suppress gene b , whereas either B or C (OR relation) can bind upstream to suppress gene a . A has the dual function of repressing gene c , or binding to D and repressing gene b .

Credit for image: Hidde de Jong [95]

are typically present in such vast numbers that they produce a continuous behaviour. The change in molecule concentrations in presence of a catalysing protein can be dramatic, producing a fast transition between states where many molecules of a molecular species are present, to ones where practically none are. When proteins bind to DNA they can in turn affect the production of another protein, thus creating a control network with genes being turned on and off, and almost discrete transitions between protein levels.

The suppression and activation of protein synthesis from a gene by other genes can be viewed as a network, with nodes being individual genes, and edges being present when a protein produced by the synthesis of some gene either suppresses or activates the synthesis of other genes. These networks are known as “genetic regulatory networks”. Figure 4.3 shows how interactions of DNA and molecules can be viewed as a network, and 4.4 shows an actual network reverse engineered from bacteria.

There are several methods for computational modelling of genetic regulatory networks [95, 149]. The most common are:

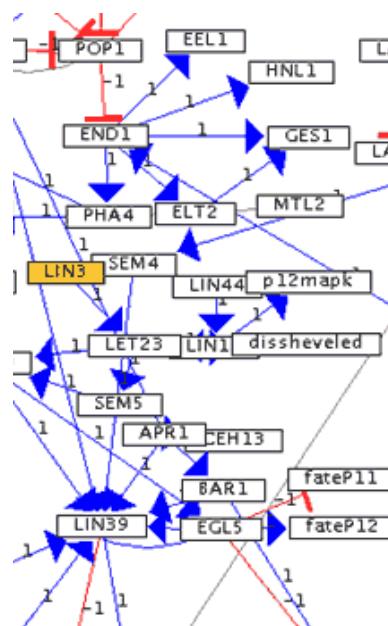


Figure 4.4: An example genetic regulatory network. Each node represents a gene which codes for a particular molecular species, with edges defining interactions between these molecules. Each edge is labelled with the effect the source molecule has on production of the target; either positive (excitatory), or negative (inhibitory).

Credit for image: BION Institute, Serbia

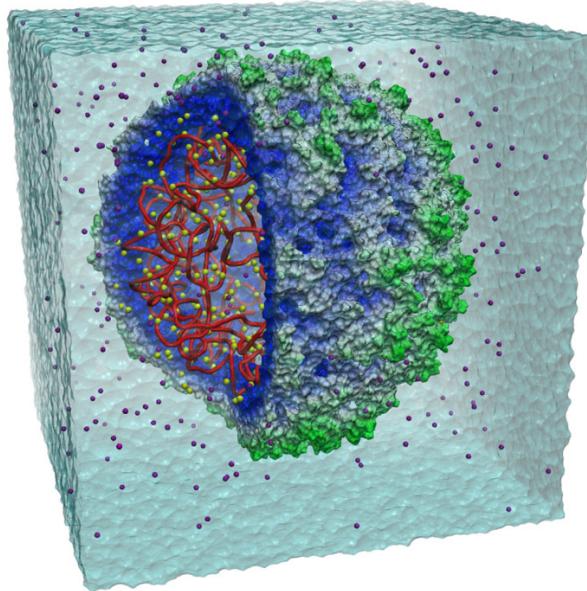


Figure 4.5: A 3D rendering of a molecule level simulation of the satellite tobacco mosaic virus. Up to 1 million atoms were simulated for over 50 ns. Stochastic simulation allows the precise location of individual molecules to be plotted.

Credit for image: Anton Arkhipov, University of Illinois [139]

- Stochastic simulation of individual molecules. This is computationally intensive as it is necessary to simulate millions of molecules within a cell in some level of detail. This type of simulation tends to be more accurate when the molecular count and threshold response levels of genes are low. Simulating individual molecules becomes computationally intractable as the number of molecules increases.

Image 4.5 shows a rendering of the first molecule level simulation of a complete life form performed in 2006, consisting of up to 1 million atoms over a virtual time period of 50 ns [139].

- A linked system of ordinary differential equations. These equations represent how the changes in levels of different molecule types are linked. The simulation must be run using some differential integrator such as the Runge-Kutta algorithm. This type of simulation tends to be used when there are large numbers of molecules, and when small variations in the levels of different molecular species do not cause significant deviations in the simulation results.
- Hybrid simulations consist of both molecular stochastic simulation and differential equation based simulation. The two types of simulation can be used as

necessary to accurately simulate a genetic regulatory network that consists of both genes that respond to low molecule counts, and genes that react on a much larger scale, in a computationally tractive way. The open source software E-cell is a good example of a system that enables distributed hybrid simulation [421].

- By modelling the network using *process algebra*, a formal technique developed for modelling synchronising, parallel processes in theoretical computer science [71]. Process algebra has also been used for modelling delay insensitive asynchronous circuits [84].
- By the abstraction of the network to a boolean network model which can be easily simulated on a digital computer. See section 4.5.

Creating network models of real world systems requires establishing the topology and derivative function of each variable through a process of experimentation and analysis. Often this is carried out by understanding the physical aspects of the system, but with the increasing size of networks and data collection, particularly in the bioinformatics field, automated techniques have been developed [36, 106], including ones that evolve models using genetic algorithms [277].

4.5 Boolean networks

Boolean networks were proposed by Kauffman in 1968 as an abstraction for modelling genetic regulatory networks [223]. Each node computes a boolean function, and hence accepts a number of true/false inputs, and produces a single true/false output. The connectivity of the nodes within a network replicates the connectivity in a genetic regulatory network, and a boolean function is used to update the state of each node given its corresponding pattern of gene activation. The computed boolean function is expected to be different for each node, as different genes have different patterns of activation. Boolean networks are also known as *NK* networks, where *N* is the number of nodes, and *K* is the number of inputs to each node. Kauffman defined a “random boolean network” as being an *NK* network with randomly generated topology and functions. Figure 4.6 shows an example boolean network.

There are two major objections to the use of boolean networks for modelling genetic systems. The first is that the whole network is updated synchronously, i.e. all of the cells are updated with new output values at the same time. This produces an artificial synchronicity that is not present in biological systems [213]. Harvey, Gershenson

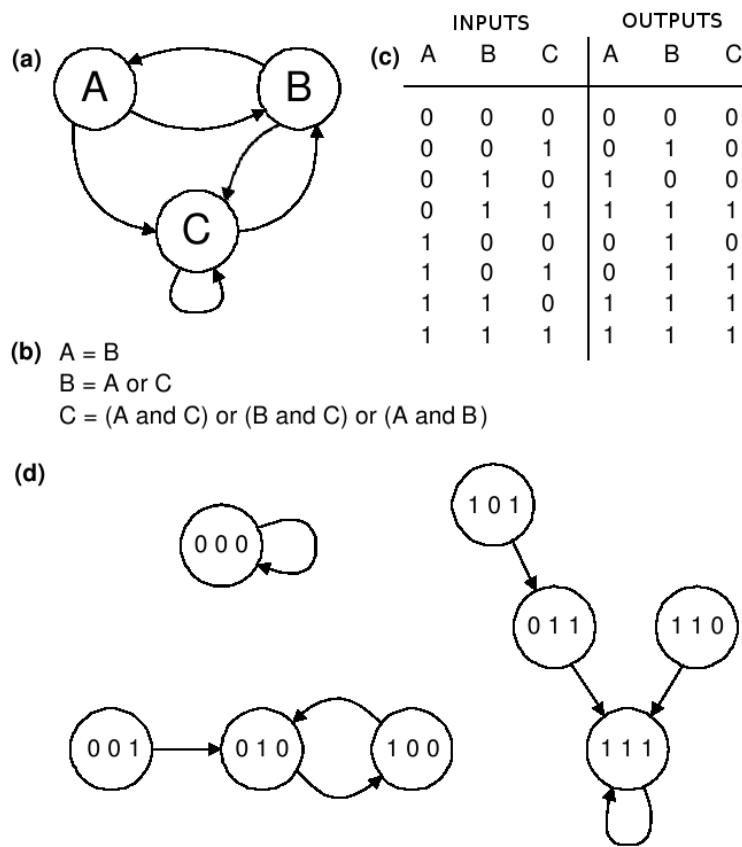


Figure 4.6: An example boolean network. a) shows the connectivity graph, b) the next state functions, c) the truth table, and d) the attractors; in this case there are two point attractors, and a single cyclic attractor. Two of the attractors have transit paths, but the 000 state is isolated.

Credit for image: N. Gead [149]

and Geard have shown that, for randomly generated boolean networks with both synchronous and asynchronous updating, attractor cycles exist which are relatively stable to perturbations, though the attractor dynamics vary greatly depending on the update scheme [150, 154, 179] (see section 4.5.1).

The second objection is that the boolean abstraction may not be appropriate for many real world genetic systems. Since concentration levels of a molecule can be so large as to be considered continuous, the control network may also react in a continuous way. It may be the case that the dynamics of the continuous system can not be accurately modelled by a simple multi-state system [371]. In contrast, Gershenson argues that the behaviour of many real-world systems is in fact determined by thresholds, such as the neural potentials governing synaptic firing, and chemical potentials governing metabolic pathway reactions [153].

Thompson has pointed out that it is the interaction of these two idealisations of synchronicity and boolean typing which is particularly deceptive, creating networks that are capable of simultaneous discontinuous changes of state that would be impossible in a continuous system [179, from personal communication cited]).

Boolean networks are an interesting target for evolutionary research as they are widely used in bioinformatics, and are considered to be accurate enough to provide further insights into the evolution of biological life.

4.5.1 Random boolean networks

A “random” boolean network is one in which both the network topology and the function of each node is randomly generated — in essence, rejecting specific topology and function as factors in favour of studying general properties about classes of network that derive from the global NK values — the number of nodes and number of inputs to each node [153]. Hence only global meta-information is considered relevant; the topology and function of nodes is not preserved across different instantiations of a particular network type. In research involving other network models, such as classical neural networks, low-level neural structures are considered essential in performing particular functions and generating particular patterns of activity. The “random boolean network” model is distinct in not attempting to preserve localised topology and function — the use of this model to explore global properties of emergence derives from Kauffman’s hypothesis that the self-organisation of biological systems is an inherent result of large, connected networks, rather than the result of some particular

micro-structures. There is some evidence from biology that network topologies are not completely pre-determined; Cherniak estimates the amount of brain-specific DNA in the human genome at around 100 megabytes — too little to encode complete information about every neuron and synapse [63]. It has been proposed that, rather than specify connectivity between individual neurons, the genome specifies the location of large scale structures, leaving lower level micro-structures to be formed using some heuristic process.

Since only global properties are considered relevant, any experimental results will apply to a particular class of network, rather than to individual networks. In a typical application, individual networks will be randomly generated from the global NK properties. Only the process of creating the network is random - once the network is generated, its topology and function will be fixed and unchanging. The network will typically be used for a single run of whatever experiment is being carried out and then discarded. Hence both topology and node function are randomised across runs, and not considered relevant factors, so any results derived from these experiments will apply to whole classes of networks, rather than some specific individual networks.

The classic Kauffman model is a network with N nodes, each of which has K inputs randomly chosen from the other nodes with uniform probability. Variations on this model are possible — the probability of an individual node being connected to another can derive from a desired statistical distribution of connectivity, such as the mean degree of connectivity of individual nodes, or some other distribution which attempts to model features of real biological networks, such as locally dense and remotely sparse connectivity — a feature also observed in genetic networks where connected genes are more likely to appear in short “canalised” sequences of DNA [472]).

Kauffman’s main research interests lay in self-organisation of biological systems. Aware of the digital abstraction being used to model genetic regulatory systems, he proposed the hypothesis that self-organisation is an inherent property of large, well connected networks [224]. He then carried out a series of experiments to determine exactly how and when digital networks display self-organising behaviours. He created so called NK networks, with a total of N nodes, each with K inputs, which were randomly connected to the outputs of other nodes. The boolean function of each node was generated randomly. The operations of these nodes were simulated to determine whether they displayed properties of self-organisation, such as the repetition of patterns caused by cyclic behaviour.

It was discovered that, at low degrees of connectivity, network activity would

quickly die out (i.e. the network would be attracted to a stable state), but around the threshold of $K = 2$, networks would spontaneously form into islands of interacting behaviour, displaying self-sustaining patterns of activation. With higher values of K , behaviour would become chaotic. Kaufman termed the area around the transition threshold the “edge of chaos” — the point at which seemingly ordered behaviour would become chaotic and unpredictable. Kauffman believed that the “edge of chaos” was an area where interesting, useful computation could occur. However, the results for high degrees of K indicate the lack of some level of biological realism, as biological neural networks have a greater mean degree of connectivity, and yet they still perform useful computation.

The self-organisation of these networks seemed remarkable given that the underlying topology was completely random. Analysis of the activation patterns showed that communication between islands was practically non-existent, but within individual islands nodes would form networks which act as dynamic attractors. The islands would have short transit periods, and then rapidly fall into their attractor cycles. The cycles were robust, with small perturbations to the state being quickly corrected.

It was found that the length and number of attractor cycles increased in line with \sqrt{N} (although this has since been challenged). Kauffman proposed that this was exactly what happens in genetic regulatory networks, with cycles of cell activity increasing linearly as the square root of the number of genes increases. Some examples of gene counts and estimated complexity of development for different species were given. However, it is easy to find counter examples of other species that do not obey this rule. Kauffman’s hypothesis has therefore not been widely accepted, but the self-organising behaviour of these networks is still a popular topic of research.

Others replicated the work of Kauffman and observed the \sqrt{N} scaling, but more recently this has been challenged (see figure 4.7). It was claimed that the number of attractors increases according to a power law [29]. This in turn was challenged, with the claim that cycle growth is in fact linear [30], and again this was challenged with the claim that cycle growth is in fact super-polynomial, and faster than any power law [375]. In 2005 this was again challenged, with the claim that growth is sub-linear. This new claim relied on the argument that biologically unrealistic attractors should be disregarded and that state space sampling is inherently biased towards reaching stable attractors, thus producing the \sqrt{N} growth previously observed [235]. The attractors considered unrealistic were those which exist due to synchronous updating, and which become unstable when the assumption of synchronicity is relaxed.

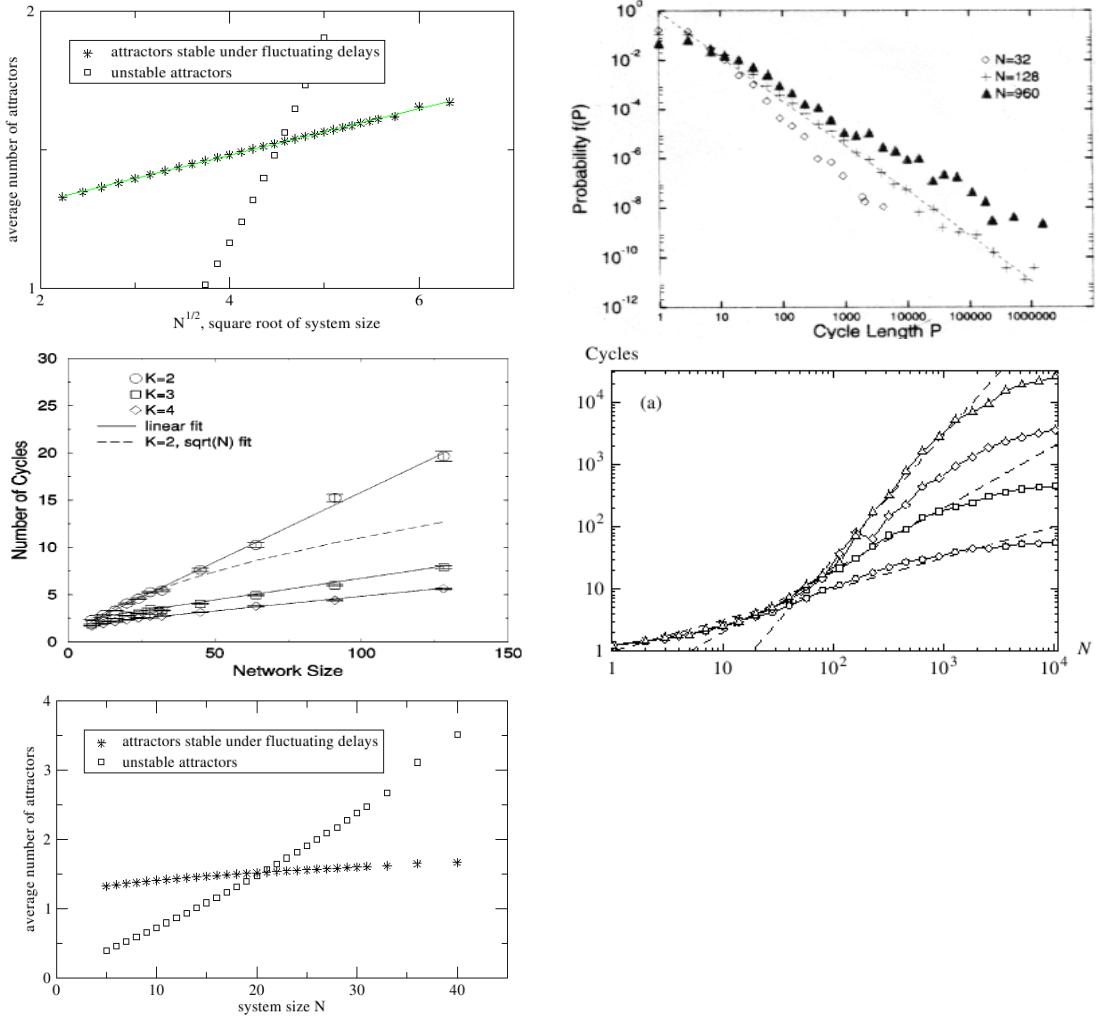


Figure 4.7: Kauffman's claim that the number of limit cycles in a boolean network grows in proportion to the square root of the number of nodes (top left) was refuted, with claims that growth follows a power law (top right), then linear (centre left), then super-polynomial (centre right), and finally sub-linear (bottom).

Credit for image: A. Bhattacharjya and S. Liang, Sven Bilke and Fredrik Sjunnesson, Björn Samuelsson and Carl Troein, Konstantin Klemm and Stefan Bornholdt [29, 30, 235, 375]

In his second book, Kauffman proposed that life itself arose as randomly connected sets of chemicals which catalyse reactions within a chemical network, thus producing a self-sustaining autocatalytic process [225] (this “autocatalytic abiogenesis” hypothesis has also been proposed by Richard Dawkins [86]). He claimed this property underlies the development of all genetic regulatory systems found in nature, and indeed, accounts for the development of cellular life from a primordial soup of interacting molecules. There is some supporting evidence for this view; in 2007 it was shown, through computer simulations, that inorganic matter could self-organise to form cell like distributions that display properties reminiscent of biological life [451], and in 2009 a series of simple, energy efficient chemical reactions that form RNA from a primordial soup were identified [422].

The attractor basin of a random boolean network can be computed by enumerating all of its possible states and state transitions (figure 4.8) [481]. This presents the obvious problem of enumerating large networks — since the number of possible states increases exponentially, a brute force enumeration becomes impractical for as few as 64 nodes. Techniques have been proposed to alleviate this problem by automatically pruning unimportant vertices, splitting the network in to sub-networks, analysing the attractor basins of the sub-networks, and then computing the global attractors by compositionally merging the sub-network attractors [116]. The amenability of boolean networks to being deconstructed in a modular fashion lends credibility to their suitability for the kind of progressive adaptation typical of genetic algorithms.

In 1997 Harvey showed that the attractor spaces of asynchronous networks are, contrary to popular opinion, radically different to that of their synchronous counterparts, with the mean number of attractors in an asynchronous network being 1 [179] (see section 6.12).

In 1998 Wuensche explored the attractor basins of random boolean networks, by enumerating all the possible states and transitions, and then visualising them. This allowed the effect of single bit perturbations to function rules to be interpreted by looking at the resulting change in attractor basins [481].

In 2000 de Paolo defined rhythmic attractors, which are cyclic attractors that move through similar, but non-identical, states, in a manner similar to the “strange attractors” of dynamical systems [107, 108] (see section 6.12).

In 2004 Gershenson compared synchronous and asynchronous update schemes and their effect on network dynamics [154]. Randomly created networks had their initial state perturbed by a single bit, and both paths were followed. After some number

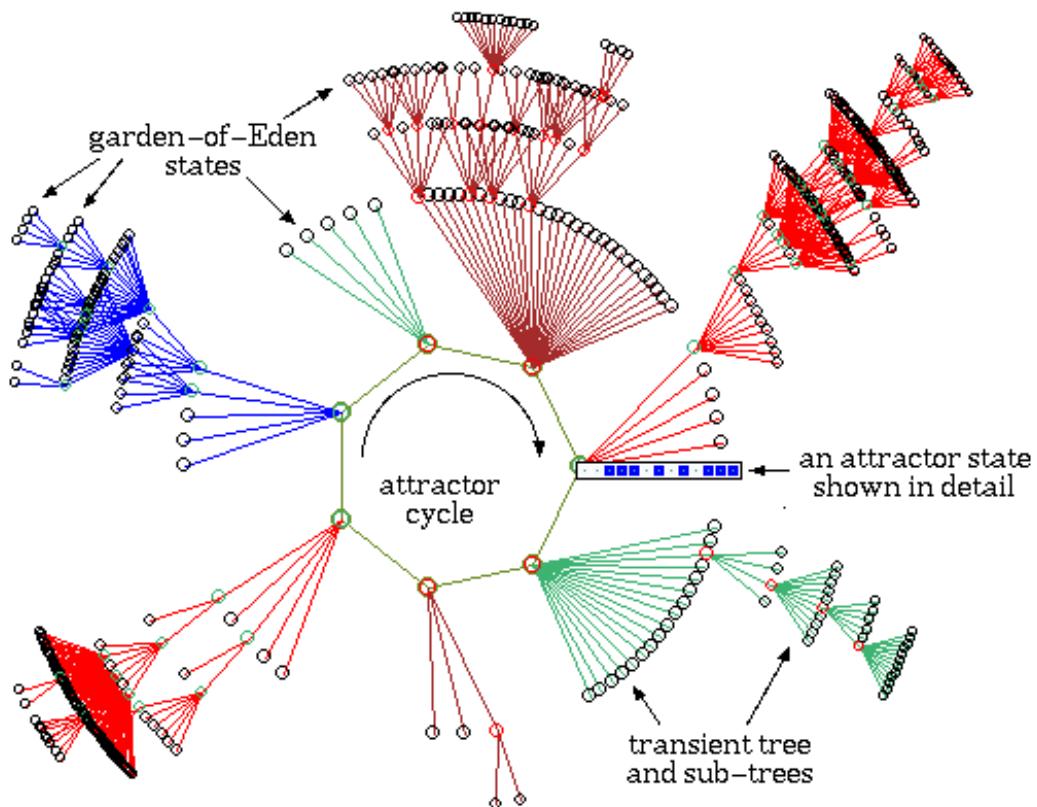


Figure 4.8: *The attractor basin of a random boolean network can be computed by enumerating all of the possible states and state transitions. A graph can then be plotted with each state as a node, and edges representing possible transitions between states. In this attractor basin initial random states quickly fall into a short limit cycle.*

Credit for image: Andy Wuensche [481]

of time steps the Hamming distance (the number of bits in the same position that are different) between the two networks could be compared, as a measure of how much they had diverged. It was shown that the choice of synchronous or asynchronous updating had no effect on the threshold stability values surrounding the “edge of chaos” (the transition region between stability and chaos, see section 4.8).

As connectivity increases both updating schemes show a change in behaviour between convergence and divergence, passing through an area where the Hamming distance stays the same, i.e. perturbations do not lead to convergence or divergence, but constantly tend towards similarly diverged networks. Both synchronous and asynchronous updating schemes could carry out “complexity reduction” by establishing attractor dynamics, and both showed a similar boundary between chaotic and stable states at some degree of connectivity K , where $1 < K < 3$.

In 2005 the effect of small perturbations on attractor stability was studied by Geard. He computed the attractor basins of various NK networks, and then examined the effect of flipping a single state bit [150]. He found that, as the network connectivity K was increased, the network response to most minor perturbations would be to remain in the same basin of attraction. Switching basins was only likely when the network connectivity was low (see figure 4.9).

From the view of genetic algorithms research these theories of the evolution of self-organising behaviour are intriguing, and research into the evolvability of NK networks for physical control tasks may provide some insight into problems of biological self-organisation. Although random boolean networks are synchronous, their stability to minor perturbations suggests a robustness in the face of other updating schemes when generating rhythmic (non-identical) cyclic patterns.

Random boolean networks, like standard boolean networks, have been criticised for their use of a global clock that can introduce artificial synchronicity. A few papers have been published on the use of asynchronous updating schemes with random boolean networks [107, 108, 179]. As there is no methodology for constructing these networks genetic algorithms were used to evolve solutions that display rhythmic cycling through similar, but not identical, patterns (see section 6.12).

4.6 Generalised logical networks

Generalised logical networks [439] are claimed to solve the two main problems with the boolean network abstraction. The first is that the dynamics of a network of boolean

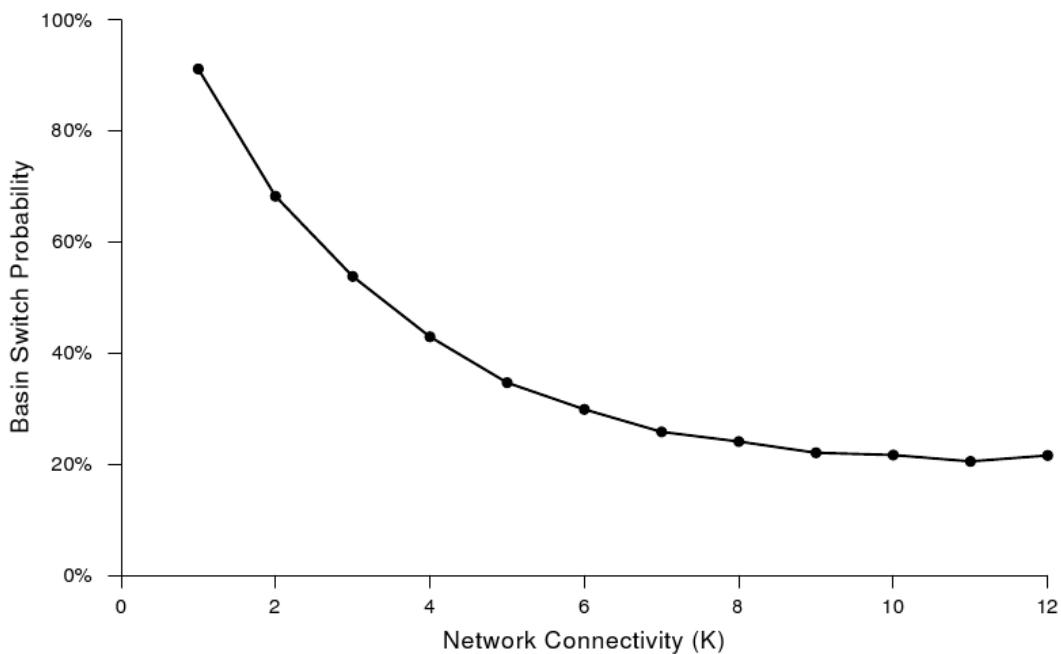


Figure 4.9: *Random boolean network attractors are often quite stable in response to small perturbations. This diagram shows how the probability of falling into a different attractor due to small perturbations falls as the network connectivity increases.*

Credit for image: Nicholas Gead [150]

functions is limited in comparison to a network of continuous functions; there are regulatory functions whose behaviour can not be reliably modelled by boolean functions [377]. To solve this, generalised logical networks use multi-value logical functions to better approximate continuity.

The second problem is that boolean networks, as proposed by Kauffman, are synchronous, which introduces an artificial source of global timing not present in natural networks. In contrast, generalised logical networks are asynchronous, with updates occurring in a random order.

Figure 4.10 shows a simple generalised logical network consisting of three genes that are mutually repressive and excitatory. There are obvious similarities to integer neural networks and multi-value cellular automata; logical equations are derived from a network with weighted edges, edges are either excitatory or inhibitory, and the logic equations approximate simple summation and segregation of the continuous states into discrete levels.

Individual variables (nodes) in a boolean network have only two states, and hence are unable to approximate the rich dynamics seen in continuous real world systems. Generalised logical systems have a multi-value approximating logic. Each variable

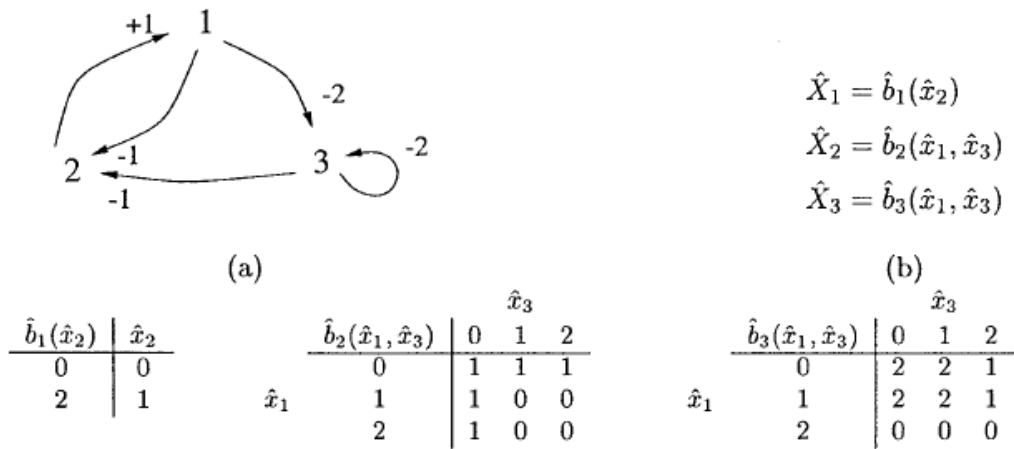


Figure 4.10: Generalised logical networks consist of multi-state nodes, representing protein concentrations, and edges, representing molecular interactions. $\hat{x}_1, \hat{x}_2, \hat{x}_3$ represent the current state of nodes 1, 2 and 3 respectively. $\hat{b}_1, \hat{b}_2, \hat{b}_3$ represent the update function applied at each node. $\hat{X}_1, \hat{X}_2, \hat{X}_3$ represent the next state of each node. (a) shows an example network, with edges labelled as excitatory or inhibitory, and a weight representing how strongly the source gene influences the target. (b) shows the multi-value logic system derived from the network.

Credit for image: Hidde de Jong [95]

can be in any one of a number of independent states. There is a one-to-one mapping between genes in a biological system, and variables in the model. The number of states for each variable is calculated by dividing the continuous concentration of protein derived from the gene into discrete, non-uniform ranges dependent on the protein's effect on other genes.

For example, a gene outputs a bounded protein concentration which can be normalised to the range $[0, 1]$. Other genes will react to different levels of this protein in different ways. If there are n unique responses, then $n - 1$ thresholds are identified which divide the continuum into n regions. The gene is then modelled as an n state variable.

The multi-value logic function for each node is derived from the genetic regulatory network, which is itself derived from real world experiments. The proteins transcribed from each gene either inhibit or excite the others (represented by $+$ /- on graph edges). A threshold between these two behaviours, taken from the discrete set of states for the source gene, is identified. These values can then be used to generate logical functions that model the observed behaviour.

Generalised logical networks have been successfully used to model and analyse biological genetic regulatory networks. One example is pattern formation in *Drosophila* [376], where signals from a distributed set of control points are integrated by the “Eve” gene to create vertical stripes along the insect’s body. Thomas used generalised logical networks to model the infection of *E. coli* cells by the λ bacteriophage [439].

Mendoza studied the cell development of the flowering plant *arabidopsis thaliana* [289]. He analysed the dynamics of the model network to find point attractors, and then experimentally confirmed that they had a one-to-one correspondence with steady states in floral and non-floral cells. Surprisingly, it was shown that there was a further steady state, which corresponded to no known cell type, and hence could not be confirmed experimentally. This state does not occur during cell development as it is not on an attractor path derived from the initial state, and it has a small attractor basin which is never reached. It is possible that this anomalous cell state is an evolutionary throwback that is no longer necessary, or it could simply be a side effect of the underlying network dynamics.



Figure 4.11: A one-dimensional CA plotted over time. Each column tracks the progression of a single node. Node state is boolean, which maps directly to a white or black pixel. Each row plots the state of the whole network in a given moment. Topology is fixed one-dimensional, directly corresponding to the layout of the row. The initial state is the top row, and final state the bottom.

Credit for image: Andrew Wuensche [482]

4.7 Cellular automata

Cellular automata (CA) are simple connectionist models which are spatially and temporally discrete [480]. Each cell has a small number of states, and interacts only with its neighbouring cells. A global update rule is defined, and at each time step the rule is applied to all cells simultaneously. All cells have identical connectivity. Cells on the edge are wrapped, forming a circular ring in one-dimension, or a torus in two. Traditional cellular automata are a subset of boolean networks in which nodes have identical update rules and uniform, geometrically regular, connectivity. Figure 4.11 shows a traditional time-state plot of an example one-dimensional cellular automata.

One way of analysing the dynamics of a cellular automata is to enumerate the state space and the transitions between different states. The states and transitions between them can then be plotted (figure 4.12). This shows the “attractor basins” — the transition dynamics by which the cellular automata moves from a large number of initial states towards a smaller group of final attractor states.

Visualising the connectivity of a cellular automata with irregular topology can (like any kind of complex graph) be difficult. One standard way supported by the cellular automata workshop software is to plot the nodes in a circle and then plot lines between connected nodes (figure 4.13).

Cellular automata are laid out in a n -dimensional space, which defines the local-

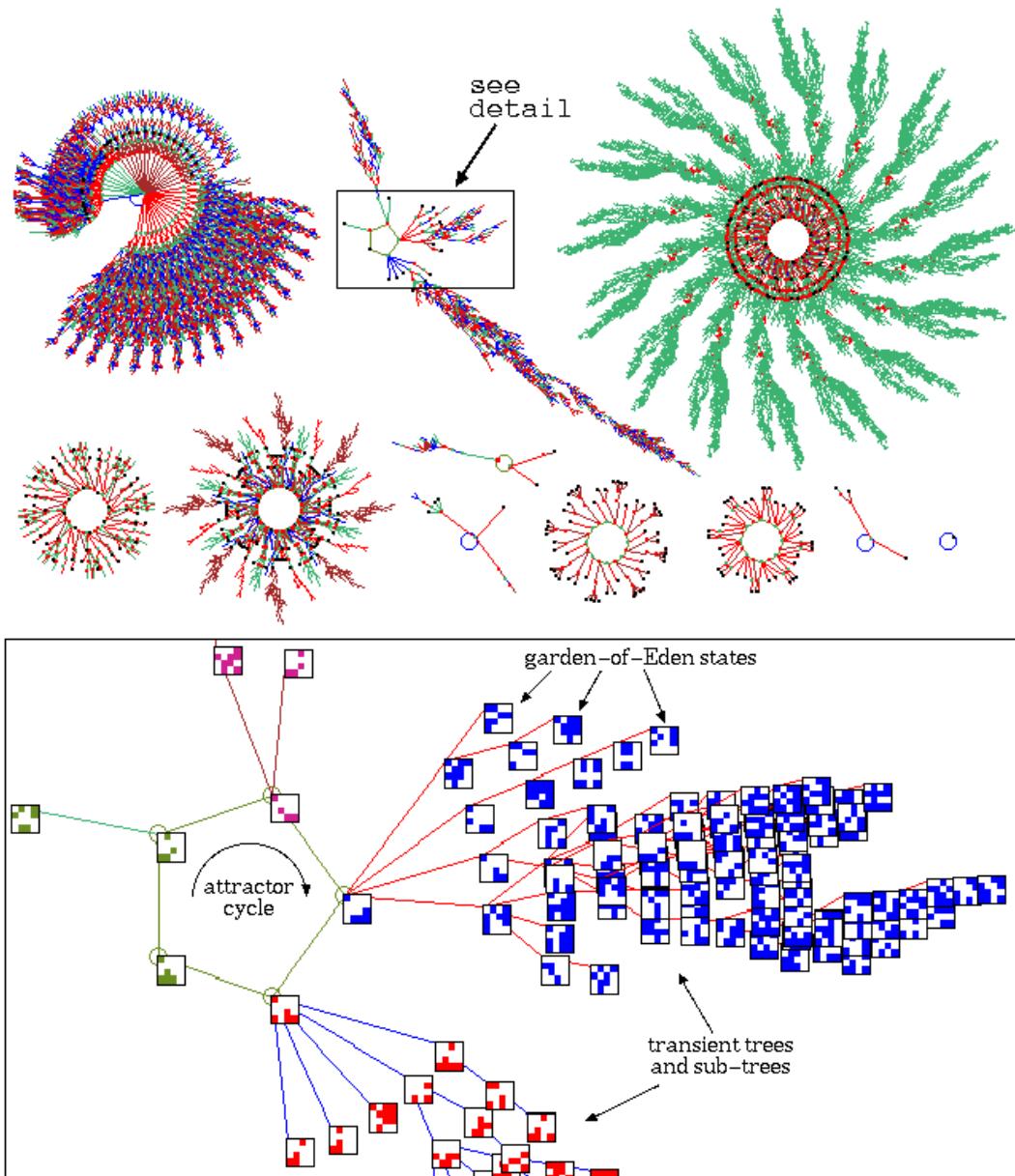


Figure 4.12: This cellular automaton contains several attractor basins. Since there are 16 cells, there are only 2^{16} possible global states, so enumeration of the states is computationally tractable. This allows all states and state transitions to be computed and plotted to visualise the attractor basins (top). The second image shows a close-up of the partial basin outlined in the first (bottom).

Credit for image: Andrew Wuensche [481]

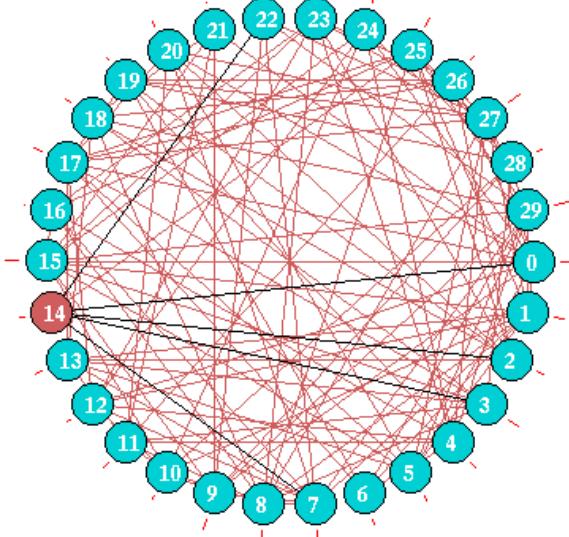


Figure 4.13: One way of viewing the connectivity of large cellular automata or other networks is to arrange the nodes into a circle and plot straight lines between connected nodes.

Credit for image: Andrew Wuensche [482]

ity of a cell to others. One-dimensional systems have been extensively studied, and are capable of showing a surprisingly rich dynamics — it has been proven by construction that a 1D system, with two states per cell, and a size 3 neighbourhood (i.e. its own state, and two incoming connections from neighbouring cells), is capable of universal computation [480, Rule 110]. The space of universal cellular automata has not been fully explored, and whilst this particular design is complex in its construction and hence unlikely to occur in the biological world, it is an intriguing possibility that similar systems could be artificially evolved to perform universal computation.

The first cellular automata to be described were synchronous, uniform in connectivity and update rule, and cells shared a discrete number of states through which they could transition. Despite these properties being defining characteristics of traditional cellular automata, it was claimed that they may make them less capable in terms of information processing. Many other forms of cellular automata have been created:

Non-uniform cellular automata (NUCA) possess irregular neighbourhood structures and individual update rules for each cell.

Structurally dynamic cellular automata allow the topology to be altered over time, as well as the cell states, enabling simulation of crystal growth, neural plasticity,

and other self-modifying systems.

Mobile cellular automata allow state and update rules to propagate between cells, and are often used to study artificial life systems.

Continuous cellular automata (also known as *coupled map lattices*) possess cells which move through continuous rather than discrete states.

Threshold cellular automata have a weighted sum-and-threshold cell update function. Essentially these are discrete, or integer, neural networks with regular uniform connectivity,

Asynchronous cellular automata update each cell independently and in a random order, and the new state becomes visible immediately to neighbouring cells [382]. Synchronous cellular automata have been criticised as being unrealistic ; in the physical world there is no global clock signal linking atoms, molecules, or cells [179, 213, 371]. Synchronous cellular automata can be constructed within asynchronous, preserving the theoretical work on phenomena like self-reproducing patterns [55], particle computation [196] and liquid computing [385], which is important as these mechanisms may underlie biological neural networks.

Partitioning cellular automata consist of synchronous sets of cells, with sets being asynchronous with respect to each other (what would be called “globally asynchronous locally synchronous” in asynchronous circuit terminology).

Non-deterministic probabilistic cellular automata cells have multiple rules that can be applied at any time step. Which one is used is determined according to some stochastic frequency distribution.

There is little difference between many of these cellular automata variants and other systems such as random boolean networks or integer neural networks, and there is a great deal of crossover between multi-state cellular automata, the systems used to simulate genetic regulatory networks, asynchronous cellular automata and asynchronous boolean networks. The defining characteristics of such systems are topology of connectivity, degree of connectivity (sometimes implied by topology), asynchronous or synchronous updating, and the number of states per cell.

4.8 The edge of chaos

The term “edge of chaos” is used to represent an area of state change in the dynamics of the observed system. Wolfram proposed that useful computation in cellular automata can only occur in the dynamical region between chaotic and stable behaviours. The reasoning is simple — a stable system has no movement of data between regions, whilst in a chaotic system data moves too quickly, and with too many unpredictable interactions between every part of the system.

A real world analogy might be to the temperature of water; when water is frozen the molecules are static, and no useful computation can occur. When water is boiling, the molecules are chaotic and move too quickly, so no useful computation can occur. In between these two phases lies the liquid state, where ripples may move through the water, or across its surface, with patterns forming as ripples flow, collide, and interact with each other, resulting in a continuous computational process.

Wolfram divided cellular automata into four different patterns of behaviour (figure 4.14) ranging from static to chaotic [480]. This system of simple classification has been criticised by Crutchfield and Hanson, who noted that the same cellular automata may display different behaviours in different regions [303].

In both cellular automata and random boolean networks it has been observed that the distribution of zeroes and ones in output rules affect the ability of the network to fall into an “edge of chaos” state and perform useful computation. Langton hypothesised that cellular automata with this property are clustered around a point in the “distribution of zeroes versus ones” space, and symmetrically around the opposite point for “ones versus zeroes” [254], although the evidence supporting this hypothesis was later discredited by Mitchell [304, 305]. Figure 4.14 shows how cellular automata classification changes as the distribution of initial rules varies.

Similar global parameters that cause computation to move through the edge of chaos have been found in neural circuits [258] and liquid state machines such as Schürmann’s mixed-mode VLSI neural networks [385].

There have been several attempts to explain how coherent systems of computation can be formed by cellular automata operating on the edge of chaos. Section 6.12 describes the evolution of particle computation systems and figure 6.14 shows an evolved cellular automaton that solves the synchronisation task [83].

The boundaries of homogeneous regions can be viewed as particles, with colliding particles producing computation forming new particles. There have been attempts to

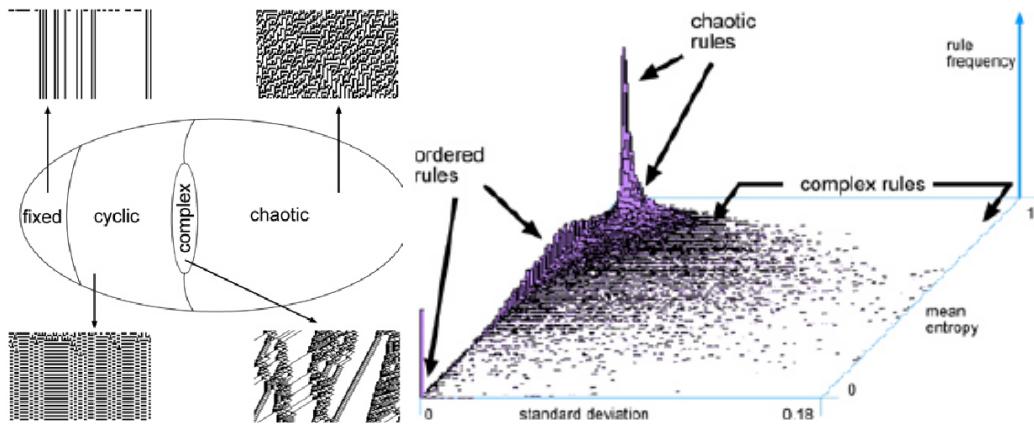


Figure 4.14: *Cellular automata can be divided into four types according to their behaviour. These are fixed, cyclic, complex and chaotic (or Wolfram's class 1, 2, 4 and 3 respectively). Global generative parameters, such as the degree of randomness in rule sets, can form thresholds that divide the solution space between these types. The 3D plot on the right shows mean entropy of cell state changes over time against the standard deviation of the entropy for a random sampling of rules from size 5 neighbourhood automata.*

Credit for image: Chris G. Langton (left) and Andrew Wuensche (right) [254, 481]

reduce these complex cellular automata to an abstract fast particle computation system based on particle classification, the probabilities of collisions between these classes producing particles of a new class, and simulation based on movement and velocity rather than numerous interacting cells [196].

Some cellular automata relying on particle computation have also been manually designed; Steiglitz implemented a ripple-carry adder [414]. “Parity rule filter” cellular automata are similar systems, using a priority based updating scheme, and are closely related to the concept of “soliton computing” in continuous systems [217].

It has been shown that similar parameter boundaries occur in recurrent neural networks, and conjectured that complex computational tasks in such networks also occur around the “edge of chaos” [28].

4.9 Summary

This thesis compares the use of complex and simpler quantised network models for some common robot control tasks. The models compared vary not just by quantisation, but also by timing and by connectivity. This chapter has provided an explanation

of the different network types used in several different fields of research, and how these network types are related by variations on common factors that will be explored by this thesis. More complex models require greater computational resources, and hence there is an interest in simpler models with non-continuous dynamics. The prospect of using digital networks for robot control is of interest, as digital systems are well understood and easier to implement. Systems like cellular automata show that interesting computation can be performed by networks with regular structure, limited communication between nodes, and simple digital update functions.

There are various different types of network, which share many commonalities. It is difficult to isolate the factors of each that exemplify their defining characteristics since there has been so much crossover in the various research areas, but broadly speaking the properties are as in table 4.1. The table shows the terms as generally understood, not what is theoretically possible, or as extended beyond the usual descriptions. For example, there are asynchronous digital circuits, and asynchronous non-uniform continuous cellular automata, but in everyday use the term “digital circuit” usually refers to a circuit with a synchronous clock, and the term “cellular automata” usually refers to a network of binary state cells with uniform connectivity and a synchronous update function.

Network	Topology	State	Update function	Update order	Signal delay
biological neural	non-uniform	continuous	sum-and-spike	parallel	dendrite and synapse dep.
synthetic neural	full, feed-forward	continuous	weighted-sum-and-sigmoid	synchronous	none
cellular automata	uniform, local	binary	uniform, boolean equation	synchronous	none
analog circuit	non-uniform	continuous	node-type dependent	parallel	geometric distance
digital circuit	non-uniform	binary	node-type-dependent	synchronous	none or geometric distance
boolean	non-uniform	binary	boolean equations	synchronous	none
random boolean	random K input	binary	boolean equations	synchronous	none
genetic regulatory	non-uniform	continuous	differential equations	synchronous	none
generalised logical	non-uniform	discrete	boolean equations	asynchronous	none

Table 4.1: *Defining features of the various network types*

Some of the concepts explored from other research areas should be of interest to practitioners attempting to model neural networks. For example, digital silicon models of neural networks to date have been implemented with synchronous circuits due to the absolute dominance of synchronous design over asynchronous within the electronics design industry. Asynchronous circuits should be considered for accurate digital modelling of biological neural systems, as biological networks have no global clock signal, hence making an asynchronous model more faithful. There are problems implementing silicon systems where individual logic components have a large fanout (where large is hundreds or thousands). Reliably propagating a single global clock signal to components that number on the scale of a modern processor is already problematic; increasing the number of clocked components up to the scale of the human brain whilst maintaining model accuracy may prove to be impossible. Therefore it is unlikely that a model of a large biological network on the scale of the human brain could be completely synchronous. Scaling up digital silicon designs to the size of such biological networks will require dropping the global clock signal, so other fields of digital design that have already explored this concept should be of increasing interest to hardware neural network implementers.

Chapter 5

Genetic evolution

So far we have examined various different types of computational network. These networks often form complex dynamical systems, making it difficult for a human designer to fashion them into performing useful activity. In order to create large networks of interacting components we abstract above the low-level details of the network by creating structured hierarchies of parts. Unfortunately the cost of this abstraction is a reduction in the computational power of individual components, turning, for example, a complex recurrent dynamical system into a simple boolean logic gate.

What we lack are methodologies for designing and reasoning about large scale dynamical systems while maintaining and exploiting their intrinsic power. However, nature, through the process of genetic evolution, has already solved this problem, resulting in a diversity of complex lifeforms displaying a range of computational and reasoning abilities.

In the design of digital circuits we do not allow arbitrary connections between nodes — every connection must be explicitly stated by the designer. We also abstract above the continuous timing model by enforcing discrete timing regimes which explicitly signal a change in time step to state holding elements. There is also an enforced uniformity in the design of nodes; each node uses a pre-determined cell design which either implements one of the logical functions AND, OR, NOT, etc. or is state holding, like a flip-flop.

In the design of neural networks we enforce that the networks are feed-forward and acyclic, and that the update function is mathematically differentiable, so that backpropagation can be used to train the network with input data sets.

In both of these cases the abstractions employed detract from the power of the underlying network. It would be desirable to employ a more advanced methodology in

order to better exploit the power of the underlying network, but unfortunately cyclic, non-uniform networks are difficult to analyse and understand. One approach that has been successful is the use of automated design techniques that do not require human analysis of the created designs. This chapter describes one such technique, “genetic algorithms”, that relies on the theories of evolution and survival of the fittest.

Genetic algorithms are used to “evolve” populations containing solutions to problems. According to the theory of natural selection, this should result in a gradual increase in the fitness of individuals within the population. This chapter will both describe those theories, and also describe how various problems can be mapped into a form which is amenable to optimisation by a genetic algorithm.

5.1 Natural selection

Evolution is the process by which superior individuals in a species survive and reproduce, whilst less successful individuals die. Although it had long been observed that children tend to inherit features from their parents, it was the English naturalist Charles Darwin who first proposed, in 1859, that species evolve over time, and that this evolution is driven by natural selection [82].

During his travels Darwin had spent 5 years aboard the HMS Beagle, tasked with studying the creatures and plant life of South America. He observed that creatures living on each of the Galápagos Islands were uniquely adapted to local environmental conditions. This led him to propose that these traits increased the probability of surviving and producing offspring on each island, and that those offspring would be likely to inherit the same traits, producing a lineage of inheritance. Initial development of these traits would be driven by random mutations that occur in the reproductive process. The reproduction of successful individuals, to the detriment of unfit individuals, would cause good traits to be passed through the generations and to be eventually distributed throughout the population.

Genetic evolution is credited with producing the diversity and complexity of all life on this planet. Over a time span of roughly 3.5 billion years life has evolved from single cell organisms to modern day plants and animals [214]. Although it has taken a long time, evolution has proved to be an incredibly powerful technique — the existence of beings as intelligent as humans is testament to this.

Humans have been using evolution to grow plants and animals with desirable characteristics for a long time; in fact, it has been known for thousands of years that individ-

ual characteristics can be passed from parents to their children. Amongst other things, selective breeding has been used to enhance characteristics such as speed (in horses), disease resistance (in wheat), amount of fruit borne (in trees), colour (of flowers) and appearance (of canine breeds and other show animals). Even the bible mentions selective breeding; Genesis 30:25 — 43 tells the story of how Jacob convinced Laban to give him all of the sheep, cattle and goats with brown streaks on their fur, and how he then arranged the watering troughs so that Laban's pure white female livestock would be impregnated by his brown streaked ones, and how he purposefully chose the stronger and healthier livestock, in order to produce strong, healthy, streaked offspring that he would also own.

In the majority of selective breeding cases it is unlikely that the human directed evolution would have occurred naturally, since development is targeted at creating human appealable characteristics which do not promote survival, such as the accentuation of visual features. This can provide an unnatural evolutionary path leading to a population which would be unable to survive in the wild. One example of this is modern wheat, which is no longer capable of seed dispersal, instead relying on human farmers to carry out this task [403].

Another example is the selection for large heads among British mastiff bulldogs, which is considered a desirable characteristic amongst show goers [370]. The evolutionary pressure towards larger heads has led to the situation where the heads of unborn pedigree pups are now too large for natural birth, and hence they must be delivered by Cesarean section. This may seem foolish, but it is an impressive display of the power of evolution; the species has been able to adapt and survive in a much changed environment, where the evolutionary pressure has changed from the ability to hunt and procreate, to simply being formed in an aesthetically pleasing way to humans (i.e. with a large head), and future generations will inherit the properties which help them survive on this unnatural evolutionary path.

5.2 Genetics in nature

In 1866 an Augustinian abbot named Gregor Johann Mendel discovered that independent characteristics such as height and smoothness of seed would be reliably inherited by the offspring of pea plants [288]. He hypothesised that certain “atoms of inheritance” were dominant, and his experimental evidence with pea plants supported this. For example, Mendel found that a tall pea plant and a short pea plant would always

produce tall offspring, suggesting that tallness was a dominant “atom of inheritance”. Although the science of genetics had yet to be discovered, Mendel was the first to propose the existence of what we now know as genes.

The cells of all animal and plant life contain a genetic sequence of deoxyribonucleic acid (DNA) [4]. This sequence contains the genetic code from which every cell in the body is constructed. DNA is a string of nucleotides; these are simple structures containing a base, a sugar, and a phosphate molecule that is shared between adjacent nucleotides in the DNA. There are four bases in DNA — adenine, cytosine, guanine, and thymine, abbreviated as A, C, G, and T respectively. A and T are complementary and will bond together, as will C and G. When two strands of DNA contain complementary bases they will bond together, and molecular forces will cause them to twist into the familiar double helix (figure 5.1).

Figure 5.1 shows several features. The *major groove* and *minor groove* are the names given to the alternating unequal gaps between the two twisted strands. The diameter of the strands is around 20 Ångströms (2 nanometres). The pair of strands twist through one 360° turn every 20 base pairs. A chromosome is a large strand of DNA; humans normally have 23 pairs of chromosomes. Figure 5.2 shows the 3D structure of a folded DNA molecule. The structure preserves the property that sequences of nucleotides which are close on a strand of DNA will also be close in three-dimensional space once the DNA has folded. This means that canalisation — the process by which related genes come to be close together in the DNA sequence — is preserved when the sequence folds, making it easier for related genes to be active and transcribed at the same time.

A sequence of three consecutive bases in a chromosome is known as a codon (figure 5.1). When the chromosome is decoded each codon produces either nothing, or one of 20 amino acids. Since there are four bases, there are $4^3 = 64$ possible codons, which map onto the 20 amino acids. There is therefore some redundancy, which acts as a defence against mutation. Two of the codon sequences act as start and stop signals for the production of a protein. The codons between these are transcribed and translated to make a string of amino acids, which are bound together in the same sequence as in the DNA to make a protein. The string of codons between the start and stop codon, which contains the instructions for the sequence of amino acids needed to make a protein, is known as a gene.

Ultimately genes are responsible for the physical structure of our bodies. They regulate a complex process of chemical interactions that produce a stable, self sus-

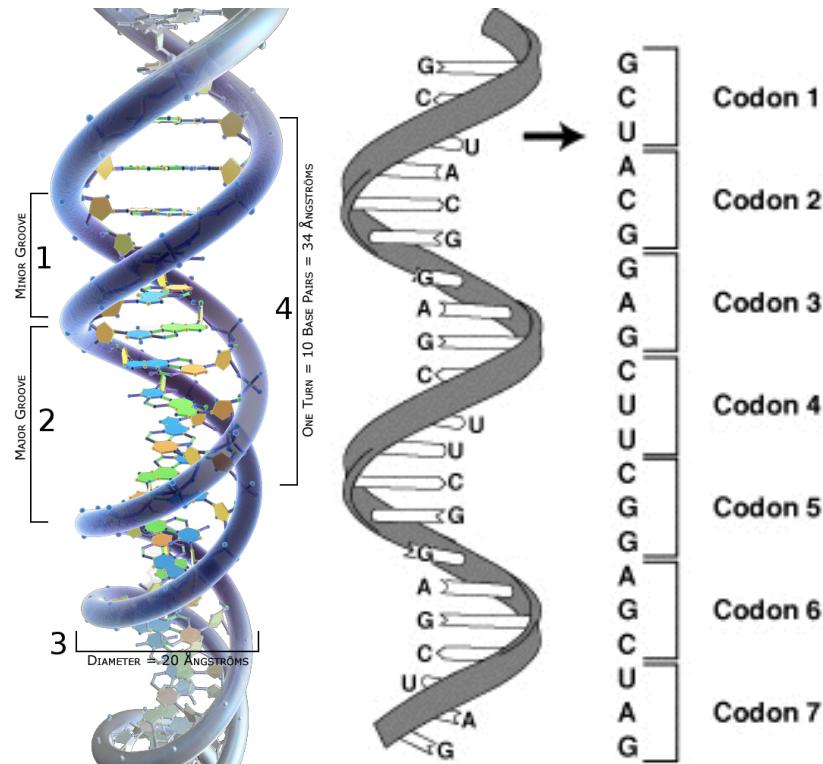


Figure 5.1: *Left: The familiar DNA double helix. Each strand consists of a backbone linking a sequence of nucleotides. The markings are labelled: 1. minor groove 2. major groove 3. Diameter = 20 Ångströms 4. One turn = 20 base pairs = 34 Ångströms. See text for explanation.*

Right: A strand of RNA contains a sequence of codons, each of which translates to a specific amino acid. The sequence can then be assembled into a protein, which will fold to form an arbitrary three-dimensional structure.

Credit for image: Zygote Media Group and the National Human Genome Research Institute

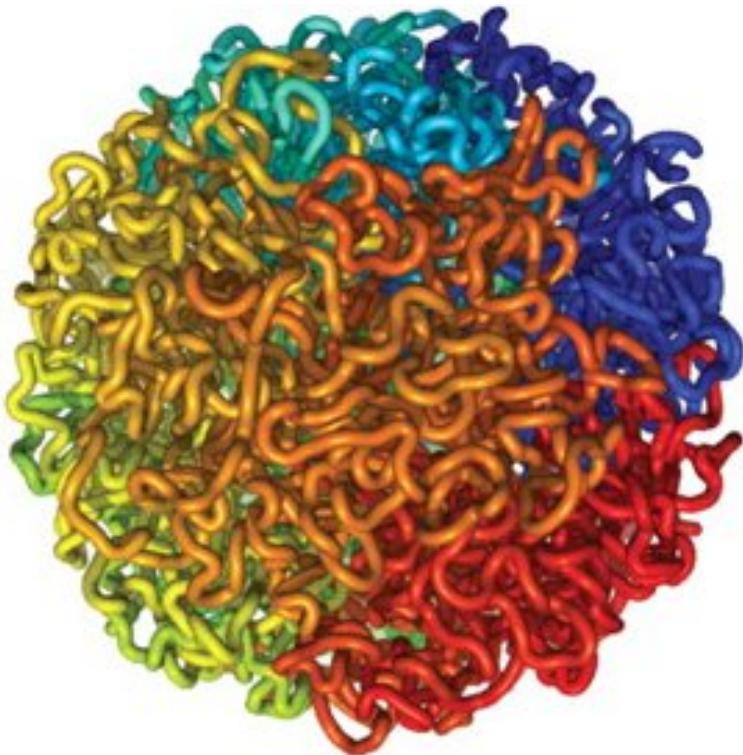


Figure 5.2: This image shows the large scale 3D structure of a DNA molecule. On a small, localised scale, a particular fragment of DNA has a double helix structure. On a larger scale, the DNA molecule twists into a structure known as a “fractal globule”, which has the property that locations that are close together on a linear DNA strand are also close together in the three dimensional structure. In this plot sequences of nucleotides are coloured similarly to show how the linear DNA strand preserves continuity and closeness when folded.

Credit for image: Erez Lieberman-Aiden et al. [259]

taining bioecology. It is this success, the complex behaviour built from small evolved components, which we hope to emulate. It is important to remember that biological evolution is an undirected process which works because of survival and reproduction of the fittest. There was no initial design plan to produce morphologies or controllers, and yet the chaotic nature of the world produces a process that is self-organising and might appear, to an external observer, to be directed by intelligent behaviour.

There are two distinct processes at work in biological evolution. When two individuals reproduce their chromosomes are combined in a process known as “chromosomal crossover” (or “genetic recombination”) (figure 5.3). To produce a child chromosome the two parent chromosomes both split into two strings. Two of these substrings (one from each parent) then join to create the child chromosome. Due to the way that the chromosomes split and the genes bond the child genome will be of similar length to the parents. This process of crossover allows children to inherit genes from both parents. Asexual reproduction, where there is only one parent, does exist in nature and is the primary form of reproduction for single celled organisms. The vast majority of multicellular species produce their offspring through sexual reproduction of two parents. This seems to be the optimal number as there are no species whose reproduction requires three or more parents.

The crossover process destroys epistasis between genes [160]. Epistasis means that genes interact; this can be clearly seen in the many interactions plotted by genetic regulatory networks. Genes that are linearly separated on a DNA molecule by a large distance are more likely to be disrupted by crossover. Hence there is an evolutionary pressure to keep highly epistatic genes close together to minimise the disruption; dependent genes which appear in such short sequences are said to be “canalised” [472].

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

Figure 5.5 shows an example of the phenotype changes that can be caused by genetic mutation that damages the “fibroblast growth factor receptor 2” (FGFR2) gene, causing Crouzon syndrome. In half of all cases, the mutated gene occurs in one of the parents and is inherited. In the other half, neither parent has the mutated gene, meaning the syndrome was caused by a new genetic mutation. This mutation is correlated with

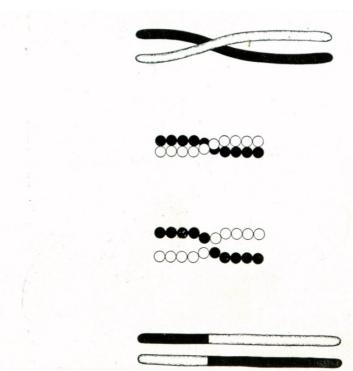


FIG. 64. Scheme to illustrate a method of crossing over of the chromosomes.

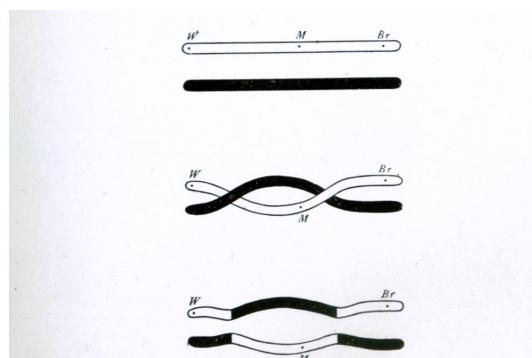


FIG. 65. Scheme to illustrate double crossing over.

Figure 5.3: *Chromosomal crossover as illustrated in 1916. Morgan had first proposed the existence of chromosomes a year earlier in his book “The mechanism of Mendelian inheritance”. The left diagram is labelled “Scheme to illustrate a method of crossing over of the chromosomes” and the right is labelled “Scheme to illustrate double crossing over”.*

Credit for image: Thomas Hunt Morgan [313]

older fathers, suggesting that it is due to damaged sperm cells, as older men have a higher percentage of abnormalities in their sperm.

Figure 5.4 shows an example of a problem during embryogenesis that has affected the phenotype — in this case, the development of conjoined twins. This may have occurred due to environmental factors in the womb, which may in turn be genetically influenced, or may be due to genetic factors being expressed in the developing phenotype. It is believed that the simultaneous release of ovaries leading to dizygotic (non-identical) twins has both a maternal hereditary component, and an environmental component, such as presence of hormones caused by diet or medication [454]. The development of monozygotic (identical) twins is not believed to have a hereditary genetic component.

The changes may result in a chromosome that performs better, worse, or the same. It may no longer perform as well when catalysing the molecular reaction that it previously operated on, but its altered shape may be able to catalyse new reactions between different sets of molecules. The evolutionary process ensures that changes for the better will survive and be propagated through the generations. In asexual reproduction mutation is the only evolutionary force introducing new genetic material; it has been shown that this is sufficient to account for the diversification of species necessary for evolution [136].

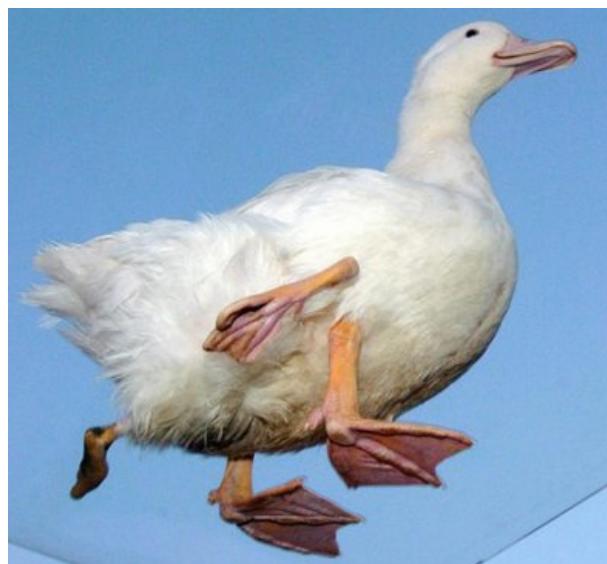


Figure 5.4: A four legged duck; this is almost certainly an example of conjoined twins, caused by problems during early phenotype development, which may be due to genetic or environmental factors.

Credit for image: BBC News [317]



Figure 5.5: An example of genetic mutation — Petero Byakotonda, who suffers from Crouzon syndrome, caused by a faulty “fibroblast growth factor receptor 2” (FGFR2) gene.

Credit for image: Extraordinary Children [131]

Mutation is an important reproductive operator; a species cannot evolve without new genetic material being introduced by mutation. There are cases where mutation brings both benefits and problems; the environment, and survival of the fittest, are the ultimate arbiter of which mutations are propagated to the next generation. One example of this compromise is the medical condition “sickle cell anemia”, in which a mutation in both parent copies of a single gene causes red blood cells to change shape [142]. It is common in sub-Saharan Africa, where it was discovered that the gene provides its carriers with greatly improved resistance to malaria. Thus evolution has shaped the human species, with geographically dispersed populations evolving different genetic traits in a delicate balance for survival.

Biological evolution is an incredibly powerful process. It is easy to see how over a long period of time it could produce species that are very advanced. What is not so obvious is that evolution works not only on the physical bodies of the creatures, but also on the way that they evolve — in a way evolution itself is an evolved process. The way in which gene sequences are converted into physical actions, and the way that chromosomes split and recombine in reproduction, are both products of evolution [324].

Speciation (in the animal kingdom) is the process by which different species are formed over time by the divergence of a single species into multiple separate species which no longer interbreed. Speciation occurs when the genomes of individuals within the species diversify in order to adapt to ecological niches. The genomes may reach the point where they are no longer compatible (for example, there may be gaps in the morphological description), or they may result in offspring which are unlikely to survive. The transfer of genetic material occurs within species, and also between species through hybridisation [273]. It has been argued that, rather than being a rare event, the landscape of ecological niches is large, that speciation is easy and likely to occur, and that distinct species, varieties of a species, and ecological races within species all represent an evolving continuum rather than a clean and discrete division of life [273]

5.3 Genetic algorithms

The study of genetic algorithms is an attempt to transfer the success of biological evolution into the domain of computer based algorithms and software [161]. Although computer simulations of evolutionary processes had been carried out since the 1950s [14], the modern genetic algorithm with multi-generational crossover and mutation was introduced and popularised by Holland in 1975 [194]. The form of the genetic

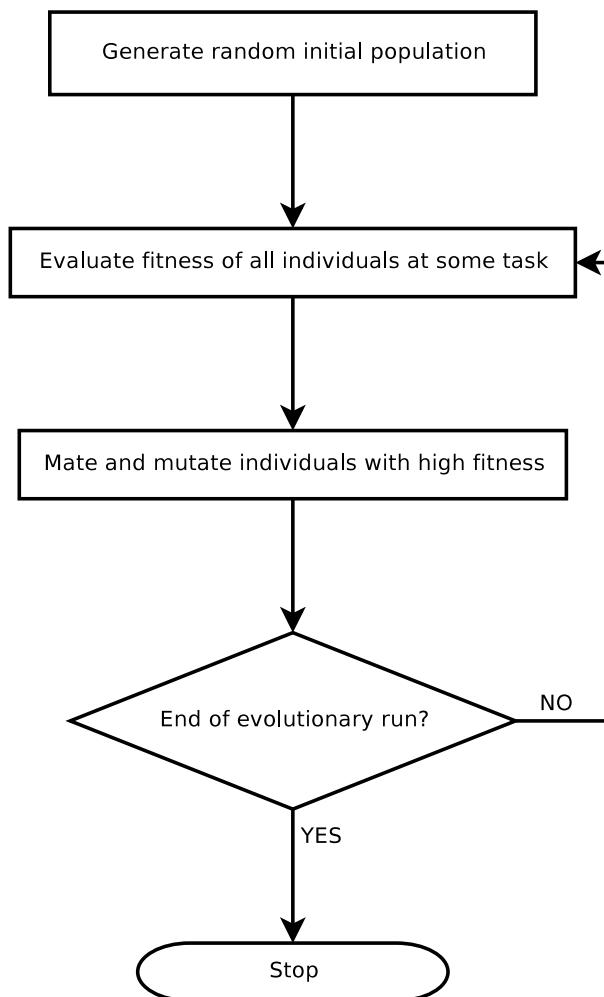


Figure 5.6: *Cycle of a typical genetic algorithm run*

algorithm has remained unchanged since then, with most of the following research focused on applying the genetic algorithm as an optimisation technique for different problems.

The genetic algorithm approach is simple — define a mapping between bit strings (the chromosomes) and potential solutions, and then, starting from an initial population of chromosomes, evaluate each one and combine the best probabilistically to produce the next generation (figure 5.6). Hopefully, combining good solutions will lead to better solutions, which will in turn be propagated through the population as happens in biological evolution. Eventually all of the members of a population will converge to a point where they have much genetic material in common and the solutions they produce are very close to each other in the solution space.

The genetic algorithm technique borrows the ideas of incremental search and build-

ing towards a good solution from biological evolution. The problem description will include an explicit function that has to be optimised, known as the fitness function. This is different from the real world, and artificial life research, where there is an implicit fitness function of survival and reproduction. Genetic algorithms are good for finding or fine tuning parameters which interact in complex and nonlinear ways.

To simulate biological evolution completely would be an impossible challenge. Nothing could be assumed about the operation of the system; random search must discover auto-catalytic sets of molecules, then discover how to combine them into a form of data that can be both copied and interpreted, thus producing a self-sustaining, self-replicating system of chemicals surrounded by an isolating wall of linked molecules which separate the internal and external environment. The computational resources required would be immense; consider a simulation of the parallel interactions between all of the molecules in an ocean. Any artificial system of evolution must be computationally tractable, and hence must abstract the true process of evolution. Recognition must also be made of the timescale of biological evolution, which occurred over billions of years. Our limited life span and research grants do not allow us this luxury.

Typically the problem that is to be solved will have a large search space, as problems within a small space can usually be solved by enumerating and testing every possible solution. The size of a tractable enumerable search space is dependent on the complexity of the fitness evaluation process. For an average process, solution spaces are tractable up to around 64 bits; in recent years it has become common for groups of individuals with access to large amounts of computer power around the globe to combine their efforts via the Internet to “brute-force” the solutions to cryptographic challenges, and 64-bit keys for symmetric encryption represent the boundary of what is possible today [453].

5.4 Modularity of the genome

Evolution is successful because smaller parts of a chromosome can be combined to produce bigger parts which preserve the functionality of their components. As populations evolve useful genes become widespread, whilst bad ones tend to die out. It has been argued that sets of genes evolve together to create functional modules, in which the behaviour of individual molecules will make little sense in isolation, but must be considered as part of a larger function [178]. Over time, evolutionary pressure on such a tight coupling between genes due to the destructive effects of crossover will result in

their relocation into short canalised DNA segments.

The reproduction process of crossover destroys relationships between distant interacting genes as the crossover point is more likely to be between them. Offspring will not contain all of the good interacting genes, so they will perform poorly and will not be selected for reproduction. Chromosomes which have the interacting genes together in a close sequence are less likely to have them broken up by crossover, and so their children are more likely to perform well and be themselves chosen for reproduction. This causes the propagation of short sequences of good genes throughout the population.

These segments can be seen as building blocks, containing sets of high performance genes, which can in turn be used as larger building blocks. Goldberg's building block hypothesis states that the success of genetic algorithms can indeed be attributed to the creation of high fitness individuals through the composition of short, high fitness schemata [137, 161]. These short, high fitness schemata, are said to quickly sample large parts of the search space, so solutions rapidly converge about them.

The building block hypothesis also helps to illustrate the type of problems that evolution is good at solving. Problems which can be broken down into a hierarchy of smaller components, which can then be incrementally solved and combined to create solutions for the larger problems, are particularly appropriate for a genetic algorithm. Neural networks are a good example, as they enable a global function of arbitrary complexity and accuracy to be modelled as the aggregate of many simpler non-linear functions.

5.5 Fitness function

The fitness function can be any evaluation of the solution which judges how good it is at any given task. The function is independent of the actual search function — it does not rely on any variables of the search process, or on knowledge of the search space around the evaluated point, which differentiates it from techniques such as gradient directed search. The fitness function does not have to be able to determine anything about the solution space surrounding a given point. This “black box” treatment of the fitness function makes genetic algorithms suitable for many problems where traditional search techniques would be inappropriate. It also introduces new problems, however, as problem specific information which could accelerate the search is ignored.

Genetic algorithms are only suitable for problems which have a number of poten-

tial solutions with a wide range of possible fitness values. A problem like searching for a cryptographic key for a *secure* algorithm would be unsuitable as the fitness surface would be completely flat apart from the single point solution which is correct.¹ Genetic algorithms have been successfully used to evolve neural networks. Similar neural networks tend to have similar dynamics, and hence display a similar behaviour. Neural networks also tend to be distributed and decentralised, and robust to noise and error. These factors may make neural networks more amenable to being evolved than some other computational systems that are more rigid and less robust, such as microprocessor code.

The choice of fitness function will have a great effect on the performance of the genetic algorithm. Several strategies have been devised that attempt to balance depth of search with exploration of new areas in the solution space. These include co-evolution, shared sampling, competitive fitness functions, and resource-sharing fitness functions [476]. Co-evolution evolves the population of fitness functions along with the population of solutions. Shared sampling chooses fitness functions that are unsolvable by as many individuals in the population as possible. Competitive fitness functions determine that a solution is better if it solves more fitness tasks than its competitors fail on than vice versa. Resource-sharing weights the credit from solving a task by the number of solutions that fail it, in order to better reward the solving of more difficult tasks.

5.6 Fitness surface

The fitness surface is defined as the dimension of fitness values of individual genotypes plotted against the dimensions of solution variables. Typically this is only useful for a small number of variables, as requires enumerating the solution space through the fitness function, and because it is difficult to plot and visualise in more than three dimensions. Figure 5.7 shows an example genetic algorithm fitness surface from a two variable genotype.

In some cases it may be possible to use principal components analysis and other dimensionality reduction techniques to enable the creation of a visualisation that cap-

¹Although genetic algorithms have been successfully used to analyse toy or weak encryption functions [26, 57], by devising a fitness function that can produce a gradiated result from the available ciphertext, it should be emphasised that this will only work with very weak encryption (with strong encryption, ciphertext is statistically indistinguishable from random data) and that there are already numerous non-genetic approaches to decrypting weakly encrypted ciphertext.

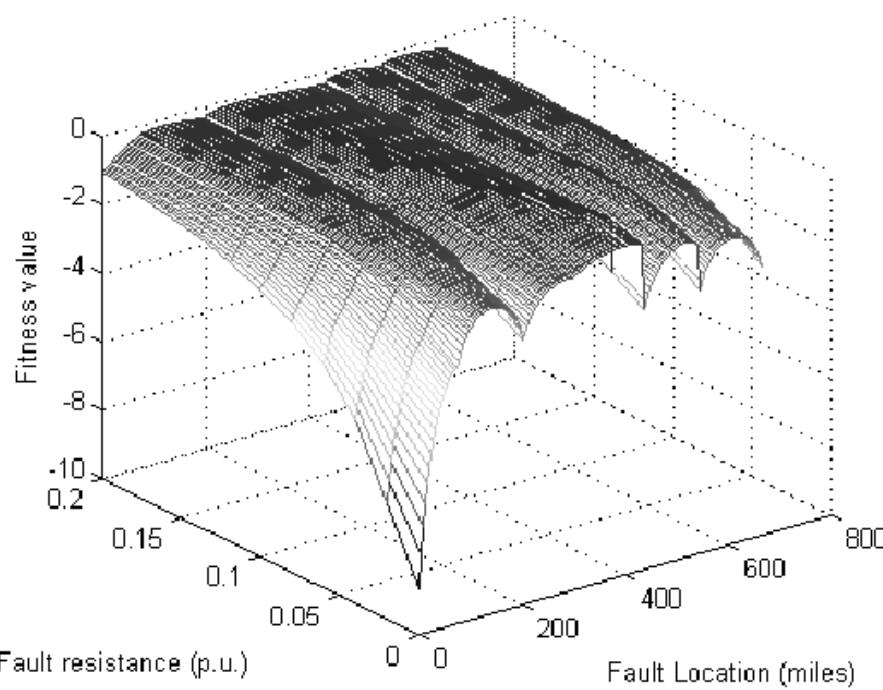


Figure 5.7: An example fitness surface. In this real world application a genetic algorithm is used to optimise fault detection in transmission systems. The population explores this fitness surface as a function of its search.

Credit for image: S. Luo [268]

tures the essence of the multi-dimensional solution (figure 5.8), but this technique does not seem to be widely used within the genetic algorithm research community, perhaps because it is difficult to extract meaningful plots when so much information has been removed. Note that creating such a visualisation would, as in lower dimensionality cases, still require enumerating parts of the solution space through the fitness function. This is part of the standard evaluation phase of a genetic algorithm, so the fitness data could simply be gathered at this stage. With a large solution space enumerating all possible solutions becomes computationally intractable, but it is still possible to sample the space using data gathered from fitness tests of the evolved individuals — principal component analysis does not require a complete enumeration of the search space.

The evolution process can be viewed as a movement of clustered points over the fitness surface. Initially the points are randomly distributed over the surface. The shape of the surface is unknown — it may be flat, undulating, have sharp peaks, or not. For the purposes of this description we will assume that the surface is one that will probably be explorable by a genetic algorithm — generally relatively flat and of low fitness, with a few hills where the fitness rises. As new generations are produced, clusters of points

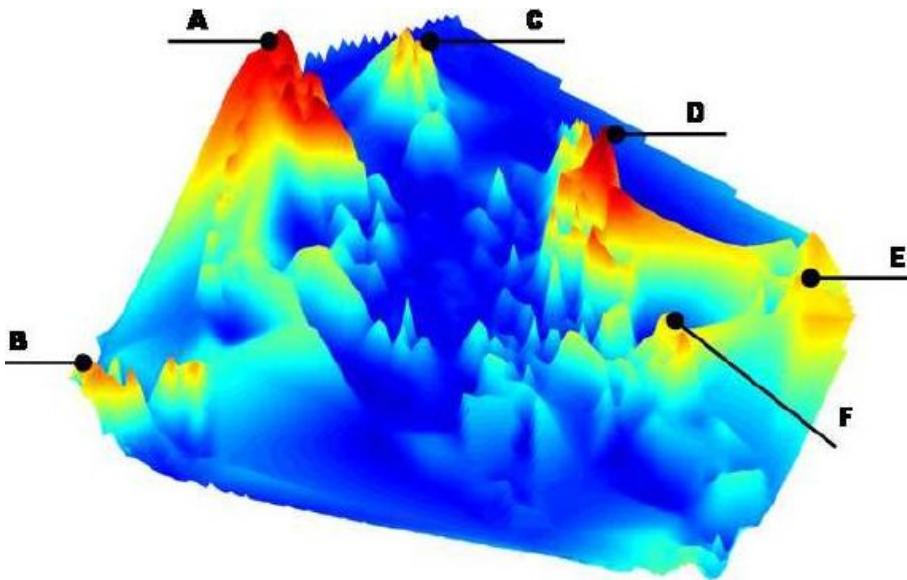


Figure 5.8: This fitness surface was produced from a “principal components analysis” (PCA) of the search space to reduce the genome to 2D points, with fitness being the third dimension. Labels indicate fitness peaks.

Credit for image: Laurent Bonnasse-Gahot [41]

will develop around hills on the surface, and they will begin to ascend them. The individuals on the flat, low fitness planes will die off. We desire the clusters to be tightly focused so that they can climb the hills, and yet at the same time widespread enough that they do not get stuck in local maxima. The evolution should be able to leap over lower fitness areas of the surface to migrate between nearby peaks. When a flat plane is encountered the population should spread out over it searching for paths to higher fitness areas.

High fitness solutions that do not converge are desirable as it indicates that the population has discovered a large plateau in the fitness surface. Children can be genetically diverse and still perform well. Within any population there is a variance in ability between individuals, and this includes the human species — although human beings are usually regarded as possessing a high evolutionary fitness relative to other large animals, there is still a wide range of ability and skill between individual humans that only mildly affects their ability to reproduce. This “fitness plateau” means that human beings still have the capacity to evolve; scientists studying the human genome have traced significant genetic changes to events that occurred within the last 5,000 to 15,000 years [467], and it has been shown that, rather than slowing down as many people believe, human evolution has actually accelerated over the last 40,000 years [183].

5.7 Search behaviour

A genetic algorithm holds a store of several points in the solution space that make up the population. The search proceeds from all of these points, rather than just a single point, with new individuals being created that are likely to be close to their parent points. This produces a “beam search” type behaviour (so named because of the searching around parallel solution points at each step of the search) [342]. Initially solutions are randomly distributed throughout the solution space. Many of these solutions will perform very poorly and be quickly removed from the population. The others will continue the search from many parallel points. As the search progresses, the fitness represented by the search space that the current population samples will tend to increase. When viewed as motion across a fitness surface, it appears that subsequent solutions are scaling the surface, ascending peaks and ridges, working towards maximal areas of the solution space. This may lead to local maxima, with one or more solutions ascending and converging at the same points, and their children failing to move away from these points. In the absence of local optima, solutions will tend to converge towards the global maxima.

Any solutions that appear to be similar are likely to be close in the search space. For example, humans share an estimated 96% of their genes with chimpanzees, and a great number of similarities are evident, both in terms of physical appearance, and internally, in the distribution and function of organs and cells [459]. The search proceeds in clusters around these points, and in new points created by combining them. As new generations are created the search can be visualised as a beam moving temporally through the search space.

The search is terminated either when the members of a population have converged around an optimal point in the solution space, or when all the individuals become trapped in local optima, or trapped on flat fitness plateaus, and hence the mean fitness fails to increase over a number of generations. The individual with the highest fitness is then chosen as the solution.

The dynamics of the genetic algorithm search behaviour can cause problems; if the population converges too quickly the solution space may not have been searched thoroughly enough and areas of high fitness may have been missed. Likewise, if the population fails to converge, or fails to expand and fully explore the solution space, then solutions may be missed. This is a classic instance of the “exploration versus exploitation” trade-off in AI search algorithms.

The population may not converge at all if this problem is inappropriate for a genetic algorithm, the search space is too big, or a bad genotype representation has been chosen. If the fitness surface contains multiple peaks the population may become split between them; different groups within the population will have converged to different solution clusters around local optima.

There is always a trade-off between exploration and exploitation of knowledge about the fitness surface. Whilst it is desirable for an evolutionary run to converge quickly, speed has to be sacrificed in order to be thorough and evaluate a diverse enough group of solutions.

5.8 Genotype encoding

Genetic algorithms operate on chromosomes, which are abstract representations of the data structure being evolved, such as the structure and parameters of a neural network. An encoding scheme is defined between the genotype, which is the potential solution represented as a chromosome, and the phenotype, which is the solution itself. The encoding scheme defines how the individual parameters that define a solution to the fitness problem are represented within the chromosome string. With a genetic algorithm, all of the parameters are encoded into a single chromosome and evolved together in parallel; this is in contrast to other approaches, such as “evolution strategies” that fix most parameters, and then vary only a small number simultaneously [10].

Different encoding schemes are necessary when we are evolving solutions to different problems. For example, in a neural network we may wish to evolve only the connection weights, or we may wish to evolve other parameters such as the topology, neuron types, etc. In co-evolutionary experiments it is common to group related aspects of the final individual into a single chromosome; one example of this is the combination of body morphology and neural network controller descriptions in the evolution of virtual creatures.

A good encoding scheme has certain desirable properties. Small changes to the chromosome should produce small changes in the final solution. This allows the process of crossover to work, as the reproduction operators disrupt the genotype, and yet the child phenotype is still similar to its parents. It is also desirable for mutation to produce small changes to the genotype, as drastic changes are more likely to destroy the good genetic data from the parents than produce the localised search around them that we desire. The encoding should encourage compactness and reuse of modular com-

ponents, allowing a complex and detailed phenotype to arise from a simple genotype, as this has been shown to improve symmetry and fitness [37, 200, 242]. One way of doing this is to allow repeated sequences of phenotype to be compressed in the genotype using a sort of run-length encoding. This technique has been used successfully by several researchers to evolve creatures with neural controllers and 3D morphology (see section 6.14).

It is helpful for solutions that occupy similar areas within the phenotype space to have genetic similarities, as it is this property of the coding which guides the search. Manipulation of the chromosome by either crossover or mutation creates new chromosomes which inherit genetic similarities from their parents. The encoding must preserve these genotype similarities when they are mapped to phenotypes.

The encoding scheme can be direct and implicit, like directly placing the parameters to the fitness function in a binary string, or it can be made more complex through the addition of layers of indirection and further computation, such as treating the evolved binary string as sequences of instructions to “grow” the phenotype. The process of converting the genotype specification into an instance of a phenotype is known as “morphogenesis”. The selection of an appropriate genotype encoding and morphogenesis process is essential to the success of the genetic algorithm.

An indirect encoding will typically contain instructions which are processed to create the phenotype. These instructions are often bundled into sequences, which are executed sequentially, with each bundle (corresponding to a gene) being executed in parallel, which is somewhat biologically plausible. In the case of neural networks, indirect encodings define the presence, connectivity, and parameters of neurons, and since the encoding is indirect there will often be a one-to-many relationship between the gene representing a model neuron in the genome and groups of actual neurons in the phenotype.

In some encodings, such as Kodjabachian’s “simple geometry oriented cellular encoding” [238], genes are directly interpreted as sequences of instructions to be executed by a virtual machine that will build a network within certain geometric constraints. These indirect encodings usually rely on the definition of strict grammars that describe the language of the chromosome string. It is also possible to exploit compression as a means to allow the development of modular reusable genes; Hornby used a Lindenmayer system to describe the co-evolved morphology and controller of virtual creatures [204], and Sims utilised recurrent graphs to describe the body parts that made up his “Blockies” [397].

Hornby and Komosinski have independently compared direct and indirect encoding schemes for co-evolved morphology and controllers of 3D virtual creatures [204, 242]. Both concluded that the evolution of developmental encodings resulted in creatures that were significantly fitter than those evolved with direct encodings. Similar conclusions were drawn as to the reason for this; that the developmental encoding allows duplication of identical parts and massively increases morphological symmetry, which both contribute positively to evaluated fitness.

5.8.1 Example encodings

This section will present some example genotype encodings for the main approaches.

Direct : $W_1 W_2 W_3 W_4 W_5$

In a direct encoding the genotype and phenotype contain the same data. In this example the chromosome contains weights for the connections in a neural network. The network topology and connections must be fixed.

Direct : $W_1 W_2 \dots W_{n^2} T_1 \dots T_n B_1 \dots B_n$

This example chromosome from [355] also describes a neural network. Like the above example, each connection has an associated weight, but in this case the network topology is fully connected so there are n^2 connection weights. The rest of the chromosome defines, for each neuron, two parameters specifying a time constant and bias. This simple encoding is powerful enough to produce recurrent networks capable of controlling 3D biped walking.

Indirect : $M(1) C(3,1) D(5)$

In an indirect encoding some processing is performed on the genotype to produce the phenotype; i.e. the genotype forms a layer of indirection. In this example the chromosome consists of a short sequence of instructions and arguments : *Make* neuron 1, *Connect* neuron 3 to neuron 1, *Divide* neuron 5. These operations could be carried out on an abstract graph, or on developmental cells placed on a two-dimensional substrate. This type of encoding was used in [238].

Developmental : $Z(1,3,1)$

- $Z(a,b,c) \rightarrow X(a)Y(b,c)$
- $X(a) \rightarrow M(a)$
- $Y(b,c) \rightarrow C(b,c)$

In a developmental encoding some processing is done on the genotype before it is interpreted to produce the phenotype. Typically, this would be to take a small, compressed genotype, and expand it into a larger genotype which can then be interpreted to create the phenotype. In effect, this creates two layers of indirection. This example shows a rule set for a parametric Lindenmayer system. Starting from the seed $Z(1, 3, 1)$ we use the rewrite rules to expand the string recursively, ending with $M(1)C(3, 1)$. Interpretation of this final string depends on the phenotype; in this case the instructions defined for the indirect encoding above could be used to produce a graph based phenotype. This type of encoding was used in [198].

5.9 Population models

A population model is a way of representing a group of individuals and their relationships. The model can affect parent selection, lifetime of individuals, and the mutation operators. There are many variations of the generic population.

In the aging population model the age of individuals is recorded [156]. The age can be used to remove old individuals from the population, or to favour or discriminate against longer living solutions in the parent selection process.

Another common model is to split the population into sub-groups, and use these sub-groups to affect the mating process, usually by favouring parents from the same group, e.g. the island model [1]. It is hoped that this will allow parallel paths of evolution to occur, with only occasional cross-pollination taking place. Another technique is to use different fitness evaluation tasks for different sub-groups, in the hope that although the groups will share common traits they will specialise in whatever task they are faced with.

In a similar vein to sub-groups, family relationships can be established by tracking the heritage of individuals. This information can then be used to affect the selection process by either preventing or promoting inter-family breeding. Whilst it is generally believed that there are mechanisms in nature to discourage inbreeding (mating between first-degree relatives) due to the restrictions it places on genetic diversity, linebreeding (breeding distantly related individuals) is often used by those practicing artificial selection in an attempt to preserve the genetic traits of specific ancestors.

With sub-group or family based population models co-evolutionary genetic algorithms can be used. These allow a number of individual populations, or identified groups within the same population, to compete against each other. This causes the

evolutionary pressure on a group to be directly related to the performance of the genetic algorithm in optimising opposing groups. This is a powerful technique for producing an “arms race” between opposing groups; in effect each is faced with a fitness evaluation task, the difficulty of which increases in proportion to the ability of the individuals [320]. This property of gradually ramping up the difficulty as individuals become more adept is in stark contrast to the usual static fitness evaluation tasks, and it has been claimed that it allows the ideal evaluation function to be approximated [94] (the “ideal evaluation function” is one which can compare any pair of solutions on all the underlying objectives of a problem — not just the explicitly stated problems, but the underlying implicit objectives that they represent).

The population size is another parameter that is commonly varied. Increasing the population size usually increases the time required to run fitness evaluations, but it also enables the genetic algorithm to explore a wider search space. Although most research uses a static population size, some have investigated allowing it to vary dynamically, dependent on parameters such as the similarity of individuals in the population. It is claimed that this kind of variation allows the best of both worlds — fast, focused evolution when the population is small, and a widening of the population when necessary; e.g. to fully explore fitness plateaus [429].

5.10 Initial population creation

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

5.11 Parent selection

Parents with a high fitness are chosen to reproduce in the hope that their children will perform as well, or better, than they did. There is a trade-off between exploring the solution space and exploiting the knowledge already gained. In a typical population

we want to choose parents that are likely to produce children that are well performing but also genetically diverse.

There are many methods for selecting parents [31]. Some of the more popular ones are roulette-wheel selection, tournament selection, rank based selection, and elitism.

In roulette-wheel selection chromosomes are chosen with a probability proportional to their fitness. This is useful when the relationship between fitness values and performance is approximately linear. For a chromosome of fitness f the probability of being selected in a population of size n is:

$$\frac{f}{\sum_{i=1}^n f_i}$$

In rank based selection the chromosomes are ranked according to their fitness. The probability of being chosen is proportional to the rank rather than the fitness value. Rank based selection is useful where there is a dramatic difference between the fitness values for similar chromosomes. For a population of size n and chromosome of rank i the probability of being chosen is:

$$\frac{n+1-i}{\sum_{j=1}^n j} = \frac{2(n+1-i)}{n(n+1)}$$

In tournament selection a group of a certain size is chosen randomly from the population. The individuals within the group compete against each other, and the fittest of the group is chosen to reproduce. Tournament based selection is useful for slowing the rate of convergence. The probability of selection for a chromosome of rank i from a population of size n with size q tournaments is [12, page 173]:

$$\frac{1}{n^q}((n-i+1)^q - (n-i)^q)$$

In an elitist selection process only the top individuals from a population are chosen to reproduce. The exact number will usually be some static percentage of the total population. In the case of sexual reproduction through crossover, pairs of elites will be randomly chosen to produce offspring. In the case of mutation based reproduction, each elite will be copied and mutated to produce offspring. This will be repeated until a full generation is created.

The elites themselves may or may not survive to the next generation. One potential problem with always allowing the elites to survive is that they can come to dominate the population by being better than their immediate children. This produces a cycle

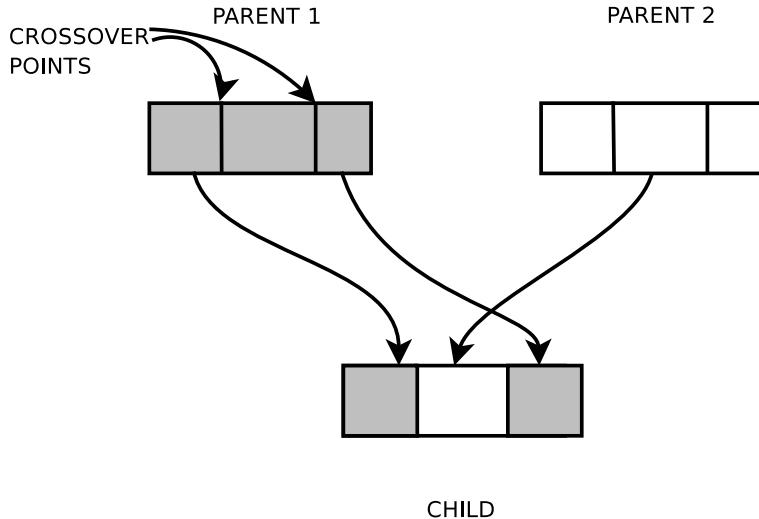


Figure 5.9: *Two parent crossover is a commonly used reproduction technique that emulates natural DNA recombination. The genotypes are represented by one-dimensional sequences of symbols. Crossover points are randomly chosen along the sequence, and the child is created by copying short sequences between the crossover points from alternating parents.*

in the genetic algorithm activity where, since the children close to the elites are not sufficiently fit enough to displace them, the same elites will be selected to reproduce in each generation. Although there may be fitter individuals on a global scale, the search becomes trapped in a cycle of local maxima.

5.12 Reproduction

The method most commonly used to combine genetic information is crossover (figure 5.9). A point in the chromosome is randomly chosen as the crossover point. Information preceding this point is taken from one parent, and the information following it from the other. These two semi-chromosomes are then concatenated to create the new child chromosome. It is suggested that, for optimal search, the probability of performing crossover when creating a child should be between 0.6 and 0.95 [11].

Mutation (figure 5.10) is used to introduce new genetic material into the population. The mutation operator is a stochastic function that somehow changes the child genotype. When using binary strings, mutation may be carried out by selecting and inverting random bits. Similarly, if the genotype consists of a sequence of symbols, the

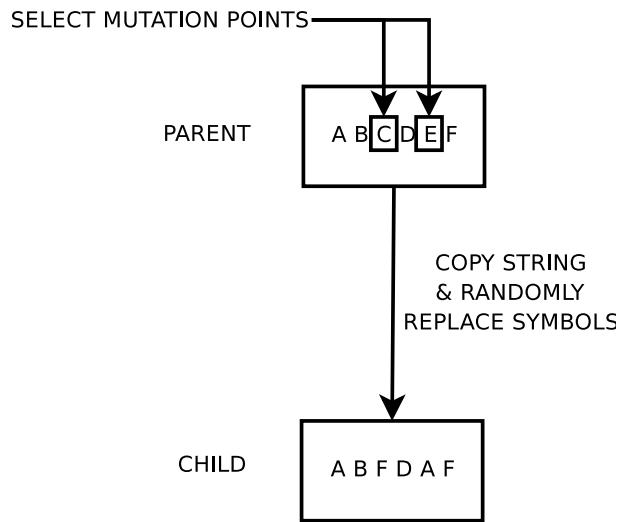


Figure 5.10: *Mutation of a genotype string of symbols.* The mutation operator can be used as the primary means of reproduction, or as a secondary operation after crossover. In both cases mutation introduces new genetic information into the genotype.

mutation operator can randomly select symbols to replace with other random symbols.

When the genotype is more complex, and must conform to some kind of structured data type, such as a graph, then the reproduction operators must be designed to preserve this structure. For a graph, crossover may take subgraphs from both of the parents, and randomly combine them, or it may copy both parents, and add edges between them. Mutation must perform meaningful random replacements or disruptions to the genotype. For a graph, this may mean adding randomly generated nodes and edges, swapping connections between randomly chosen nodes, swapping parameters of nodes, duplicating nodes and connections, or deleting nodes. Goldberg has suggested that using such higher level operators is unnecessary and suboptimal, and that instead all genotypes should be represented as bit strings, using standard crossover and mutation, regardless of the structure of the eventual phenotype [161]. In this case, crossover may occur in the middle of some parameter, corresponding to a crossover operation on either side of the encoded parameter's boundaries followed by a mutation operation on the parameter itself.

As mutations are often disastrous, and they can destroy inherited characteristics, a low mutation rate is usually chosen. Goldberg suggests a rate of 1 mutation per 1000 bits [161]. Back has suggested that the optimal mutation rate should initially be slightly

higher, but fall towards $\frac{1}{n}$ (where n is the genome length) as time progresses [11]. Thierens has suggested that the optimal mutation rate should dynamically adapt to evolutionary pressures and progress of the genetic algorithm itself [437].

5.13 Summary

This chapter has introduced genetic evolution — the biological process which is responsible for creating the various forms of life in the world, and uniquely adapting species to their environments. This process works in living cells through the act of reproduction, which is subject to genetic crossover and mutation. Crossover allows genomes from different individuals to be combined to produce a child genome, whilst mutation acts to introduce random changes to the genome. Genes are inherited, so changes which increase the ability of an individual to survive and reproduce will go on to be passed to the next generation, whilst genes that are detrimental will produce a less fit individual, who is less likely to reproduce. Over time this iterative process leads to an increase in the fitness of the population.

Certain problems, such as those containing numerous dependent variables and a large problem space, have defied classic engineering approaches. It is desirable to solve these problems using an automated approach that can treat solutions as a “black box”, where the internals of the problem solution are not subject to direct engineering, so that the need to analyse and decompose the problem in an engineering manner is eliminated. The field of genetic algorithms applies what we know about the algorithmic process of biological development to problems that can be specified as a computational process.

In order to apply a computerised genetic algorithm to a problem domain there must be some way to quickly evaluate solutions to the problem. This will usually take the form of a simulation of the real problem environment. The space of possible solutions must be encoded as a genotype that evolutionary operators can act upon. This is done in a problem specific way, and with regards to the evolutionary functions, since the coding and functions that act on it are intrinsically linked. Encoding schemes can directly specify solution parameter values, or can use developmental embryogenic schemes to “grow” the solution, in effect providing a layer of indirection that enables reuse of parameters and segments of the genome.

There are various population models used in genetic algorithms that attempt to artificially simulate geographical division, speciation, familial relationships, and con-

flict between opposing groups. There are various schemes for evaluating and selecting parents that trade-off explorative search of the solution space for exploitation of knowledge already gained about promising solutions.

Chapter 6

Evolution of specific genotypes

The last chapter described the basic processes of evolution, and how genetic algorithms utilise these processes in order to create and optimise solutions to difficult problems. This chapter will expand on the previous one by describing how the solutions to specific problems, such as the design of neural networks, cellular automata, and other network types introduced in chapter 4, can be mapped into a form usable by a genetic algorithm. Past research in the use of genetic algorithms to discover solutions to these specific problems will also be discussed.

In many cases of genetic algorithm usage we are looking to create systems that display properties of “emergence”; that is, a global behaviour becomes evident that is greater than the sum of the parts. For a behaviour to be emergent there has to be a synergistic relationship between the individual units. Some of the fields where emergent behaviour can be found include creature behaviour, such as flocking, and neural networks, where composite network behaviour is a function of many individual neurons.

One example that typifies the use of genetic algorithms to evolve emergent behaviour is in the evolution of networks of nodes which, through strictly localised interactions, produce globally recognisable patterns when run. This is a recurring research topic, and there are examples of evolved cellular automata, boolean networks, genetic regulatory networks, and neural networks, which all act as pattern generators.

Pattern generation will therefore be covered as a separate topic distinct from the generic evolution of these network types for other problems. The evolution of pattern generators is also directly relevant to this research, as it is suggested that pattern generators in biological networks are responsible for locomotion in a wide variety of invertebrates and vertebrates, including humans [117, 270].

As the problems to be solved by genetic algorithms are diverse and cover a wide range of fields some have argued that they can be more successful when utilised in a problem specific way. There are also arguments against this, so we will begin with this topic.

6.1 Problem specific operators

It is possible to introduce genetic operators that manipulate data structures inside the genotype using problem specific knowledge. The traditional crossover and mutation operators are called “genetic” because they are based on the processes that occur with DNA in biological cells. Deviating from this risks turning the genetic algorithm into a generalised search that may not display the same behaviour and properties as the desired evolutionary search. In a similar vein, the solutions do not have to be represented as binary strings, they could be graphs or use some other problem specific representation or data structure.

Biasing the search in this way requires careful consideration; eliminating large parts of the search space, or favouring particular data structures, will only shorten the search process when it is known beforehand that the optimal solutions lie here. In many problem cases, use of a genetic algorithm is considered precisely because the structure of the best solutions is not known beforehand, and in some cases what humans thought was the best structure may actually be radically different from that discovered by evolutionary search (e.g. see the antenna designs in figure 6.30).

Holland argues that, in order to achieve the benefits of genetic algorithm search, all solutions should be coded as binary strings [194], which simplifies the genetic operators, and reduces the potential for biasing the search process. However, it is clear that other codings along with corresponding operators can be equivalent to the manipulation of binary strings. The argument is somewhat irrelevant since problem specific knowledge is being used to devise a genotype to phenotype mapping that preserves similarities between both, and this mapping can itself bias the search, even when using binary strings as the underlying genotype representation.

Using problem specific operators avoids the problem of redundancy and errors when interpreting the genotype. For example, a 4-bit parameter ($0x0 \rightarrow 0xF$) may be encoded in a standard *int* data type, which is 32-bits ($0x0000 \rightarrow 0xFFFF$), by using the least significant four bits and disregarding the other 28 most significant bits. If the 28 most significant bits are simply disregarded, then every sequence of 16 values

within the 32-bit space will be mapped onto the 16 values of the 4-bit space, and this will be repeated for all of the possible 32 bit values. The 32-bit space will be uniformly divided and mapped onto the 4-bit space. However, if a ceiling function were used to map all numbers above or equal to 0xF in the 32-bit space to the actual value 0xF in the 4-bit space, then the value 0xF will become disproportionately represented in the resulting 4-bit space. Mutation works by enabling small changes in the genotype to effect non-destructive changes, which are usually also small, in the phenotype. So, even with binary strings, the genotype to phenotype mapping can still heavily bias the search, and thus requires some consideration.

It should be noted that much of the research in the field of genetic algorithms uses problem specific operators. This is often due to the use of graphs to represent a genome (e.g. network structure, morphologies) and the difficulty of finding a morphogenesis process that maps binary strings into graphs whilst preserving the property that mutations should produce phenotypes similar to the parent one. Graph specific mutation operators, such as node and edge removal or addition, or node attribute modifications, are often considered easier to implement.

Using the same strong typing for parameters stored in both the phenotype and genotype avoids problems with coding and mutating unusual data structures. In particular, the common use of floating-point variables makes mapping to and from binary strings difficult. Various integer coding schemes, such as Gray coding, have been devised to ensure that mutations in the genotype produce effects of a similar scale in the phenotype, but no such coding scheme exists for floating-point values. Using the direct binary representation is undesirable since the IEEE-754 standard for representation of floating-point numbers does not preserve the desired mutation property, and does not provide a catch-all way of converting random bit strings into valid numbers [159].

Early researchers who tried to use random bit strings found that not all sequences were valid floating-point numbers, and that the floating-point units of various processors would either throw exceptions, or generate unpredictable results. Hence it is common for floating-point values to be stored as their primitive types, with mutation either consisting of replacement with a randomly chosen value, which does not preserve the genotype mutation scale property, or replacement with a value chosen from some Gaussian distribution centred on the old value, which does.

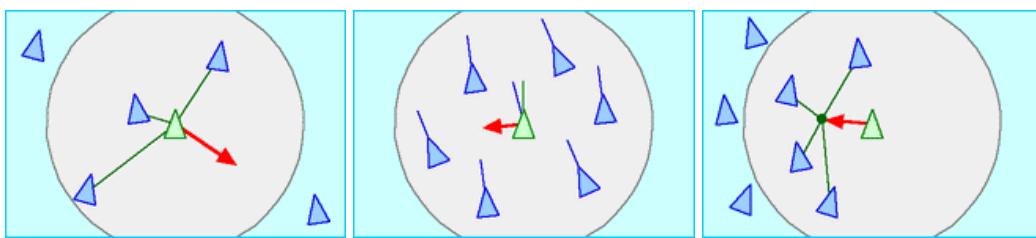


Figure 6.1: *Three simple steering behaviours can generate biologically realistic flocking behaviour. “Separation” forces agents apart (left), “alignment” forces agents to become oriented towards the same direction (centre), and “cohesion” forces agents to move together (right).*

Credit for image: Craig Reynolds [358]

6.2 Evolving flocking

In 1987 Reynolds proposed algorithms to model the flocking behaviour observed in the group movement of biological species such as birds [358]. When viewed as a global behaviour, bird flocking appears to be incredibly complicated. Each bird can only sense a small area around it, and can only control its own movement. There is no direct communication between birds; the only form of data transfer is by movement and visual sensing. Despite these local constraints bird flocks demonstrate a remarkable form of global synchronicity. A flock will tend towards the formation of a single “V” shape, with individual flocks merging to create larger ones. When faced with an obstacle, a flock will cleanly bifurcate into smaller groups, each spontaneously forming its own flock, ready to merge again once the obstacle is passed.

Reynolds was intrigued by the *emergence* of this global behaviour from the simple interactions between actors. He proposed that three simple local rules, when applied to each bird, could account for the global behaviour (figure 6.1). Computer simulation of the birds, which Reynolds termed “boids”, showed that the rules created a global emergent behaviour very similar to that of real birds.

Reynolds’s work utilised simple point mass simulations. In 1997 Brogan and Hodgins showed that similar algorithms would generate flocking behaviour in systems with complex 3D dynamics, such as realistic simulations of humans riding bicycles at speed (figure 6.2), in which all of the joints and body parts making up the human body are accurately modelled [45].

Several researchers have used genetic algorithms to evolve individual rules for agents to display a global flocking or swarming behaviour. In 2003 Trianni evolved



Figure 6.2: *Emergent flocking behaviours, such as object avoidance, were shown to be possible for composite 3D bodies with complex dynamics, such as one legged robots (left) and bicyclists (right).*

Credit for image: Brogan and Hodgins [45]

neural controllers for circular two wheeled robots [450]. The fitness function is designed to judge distance of a set of several individuals from the centre of mass, in order to reward clustering behaviour.

In 2005 Spector used the open source package Breve to create a virtual life system with flying agents (figure 6.3) [406]. Rules for flight control were encoded in the genome. Being an artificial life system rather than a generational algorithm meant that agents were subject to a continuous battle for survival. Agent actions, such as movement and reproduction, carried an energy penalty, whilst the consumption of food allowed agents to gain energy.

Spector observed an interesting evolved emergent group altruism; a few individuals would feed on the energy and live a long time, while their children, who were genetically identical, would form a swarm around the energy and quickly die. Analysis showed that the swarming children were blocking creatures from different genetic groups from approaching the energy sources. The behaviour of the short lived children swarming around the periphery of the energy region was self-sacrificing, in order to ensure survival of the group genome. This kind of behaviour is common in insects, such as bees and ants, where siblings share a greater proportion of DNA than parents and their children.

A further refinement to the system gave agents a genetically encoded colour and the sensory perception to differentiate between differently coloured agents. Agents were also given the ability to transfer energy from themselves to other specific agents. When this simulation was run, agents evolved who would use their perception to identify agents of the same race (i.e. having a similar genotype, leading to the same colour) and altruistically beam energy to them if their energy stocks were low.



Figure 6.3: *Emergent, global flocking behaviour results from the many interactions of evolved agents.*

Credit for image: Lee Spector [406]

In 2005 Stanley presented a real-time version of his “neuro-evolution of augmenting topologies” (NEAT) algorithm (see section 6.4), and demonstrated in-game evolution of swarming and fighting behaviour for multiple agents (figure 6.4) [412]. In this system neural networks are progressively evolved and used to control armed agents in a 3D environment. Fitness tasks which promote cooperation and global, as opposed to localised, strategies are used to evaluate the group performance of evolved genomes.

6.3 Ant colonies

In 1991 Dorigo published the “ant colony” optimisation algorithm [74, 112]. Again, inspiration from biology was a primary motivating factor in this research. From the perspective of an outside observer it appears that ant colonies, when viewed as a whole, display a coordinated, globally synchronised behaviour, and that individual ants are aware of overall goals and aims and select their actions accordingly. However, this is not the case, as we know that the sensory perception of ants is limited to their local neighbourhood.

Dorigo found that simple rules, when applied to interacting mobile agents, could account for the global ant behaviour. Interactions are mediated by communication between ants based on the release of scent signals, which form pathways as ants move around. Other ants are attracted to these pathways, release their own scent as they

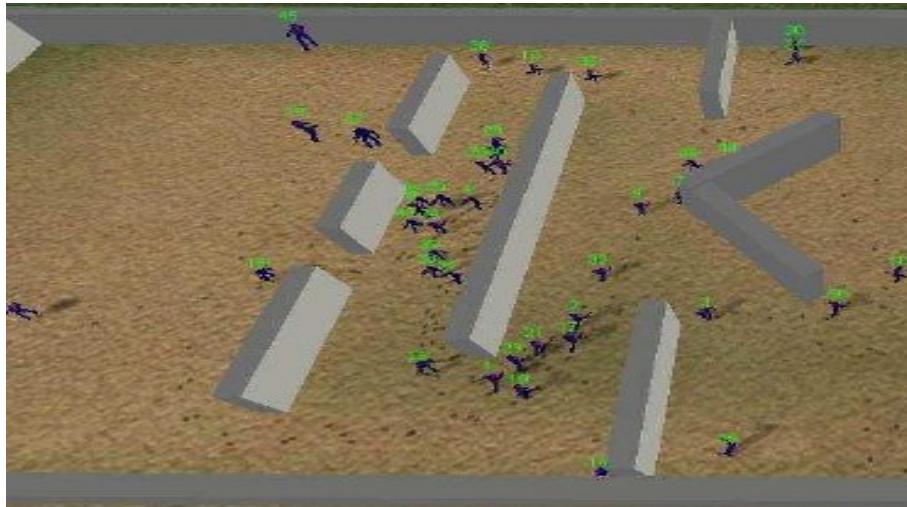


Figure 6.4: *Real-time evolution of neural networks for swarming and fighting behaviour in the Nero video game.*

Credit for image: Kenneth O. Stanley [412]

move along them, and thus reinforce them, increasing the attraction of well travelled paths. If the pathway is advantageous to the individual ants, for example leading to food, then many ants will traverse it, creating a very strong attractor. If the pathway is not advantageous, ants will be less likely to follow it, and the scent will dissipate over time, or be disrupted by the criss-crossing of other scent pathways (figure 6.5).

It was shown that rules that promote random search along with scent release and following will create a globally emergent behaviour. The observed behaviour of the simulations closely matches that of the real world when tested, e.g. wiping away part of a strong scent trail will result in random searching by the ants at the head of the trail, followed by rapid re-establishment of the broken link. This and other types of emergent behaviour appear in both the real world and Dorigo's simulations.

Buttazzo has suggested that the emergent behaviour of an ant colony is similar to the functioning of the brain, in that they both consist of large numbers of cooperating units with tightly coupled interactions, and that consciousness of the mind emerges due to the same properties that bring order to the ant colony [53]. This is in line with hypotheses that the brain consists of distinct, autonomous modules controlled through hierarchical inhibition, and the “society of mind” hypotheses from philosophy.

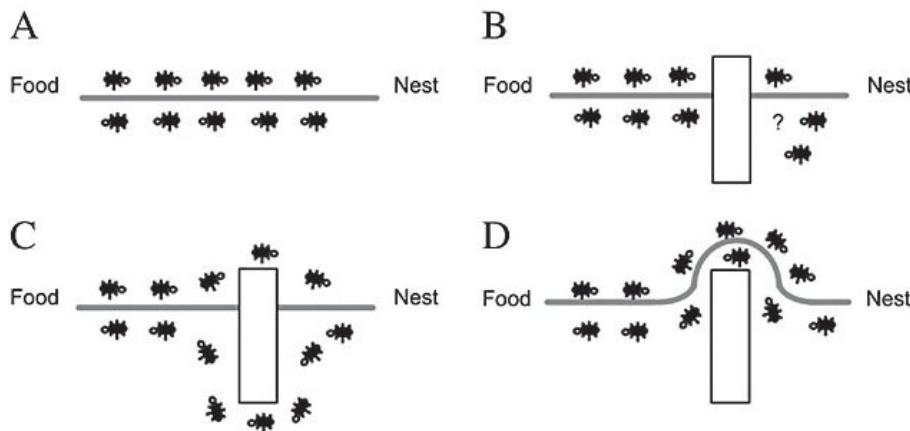


Figure 6.5: *One advantage of ant colony optimisation over similar methods, such as simulated annealing or genetic algorithms, is that it is adaptive over time. This diagram illustrates how ants establish a new pheromone path when an object is placed to block the old one. The dynamics of the attractor reinforce the shortest path.*

Credit for image: Mauricio Perretto and Heitor Silvério Lopes [266]

6.4 Evolving neural networks for robot control

Neural networks contain many parameters and variables which can be altered by a genetic algorithm. Not only can individual neurons be parameterised, but the connection weights, and connectivity of the network itself, can also be created and optimised. For surveys of genetic algorithm use to evolve neural network controllers see [236, 293, 486, 487].

It has traditionally been accepted that although genetic material is often directly transferred between the genomes of different species of plant, producing hybrids, the same is unlikely to occur between the genomes of different animal species, as by definition they are unable to reproduce together. Hence it has been widely accepted that this kind of “crossover” operator would have had little effect on animal evolution. In recent years this dogma has been challenged, and it has been recognised that hybridisation between different species does occur in nature, and has been a driver of evolution [273]. It is estimated that at least 25% of plant species and 10% of animal species, have been involved in hybridisation with other species [272].

In neural network reproduction the crossover operator is usually abandoned and mutation used as the sole operator. This is often done as there is no clear way of combining data from separate chromosomes to produce offspring that are likely to survive. In the case of neural networks, creating a child by splicing two different binary strings

together is unlikely to succeed; unless the parent networks have a similar structure the changes introduced by combining them will be the equivalent of massive mutation, which is almost certainly undesirable. Crossover works when there are identifiable functional units within both the genotype and phenotype structures, and this must be carefully considered when applied to neural networks.

Parameter values, such as connection weights, neuron time constants and biases, are represented as floating-point values which are mutated by replacing them with either a randomly generated value, or one drawn from a Gaussian distribution centred on the current value. Replacement with randomly generated values produces a wider search through the solution space, which will usually slow the search, since improvement usually occurs as individuals ascend ridges along the fitness surface, but larger steps are likely to fall beyond these ridges in low fitness regions. On the other hand, replacement with a value from the Gaussian distribution centered around the current value means children will be closer to their parents in the solution space, and therefore more likely to be close on the fitness surface, producing a narrower, more focused search.

It has been observed that as the size of a network topology increases its evolution becomes more difficult. This is due to the increasing number of weights that have to be substantially correct in order to provide any basic functionality. For evolution to succeed, it is necessary to provide a clear evolutionary path from small, simple networks, that perform reasonably well, up to larger, more complex ones, that perform better.

One approach to this is to use staged evolution, in which evolution is carried out on small networks, which are then either frozen (allowing the network to be used as a module but with no internal changes), or new neurons are added but their connection weights are deliberately very low so that they will not disrupt the current network. In these schemes network size is slowly increased over time in order to encourage the development of simple networks early on, which become more specialised and optimised over time.

Analysis of evolved neural networks is difficult since they display non-linear high-dimension dynamics. Many researchers perform no such analysis, treating the internals of evolved networks as a black box. Others manually conduct lesioning experiments, in which one neuron is selectively removed, and the effects observed [3]. This is similar to experiments carried out in biology to discover how the nervous systems of creatures control the body. The “functional contribution analysis” (FCA) [388] and “multi-lesion

Shapley value analysis” (MSA) [227] algorithms both perform automatic lesioning of multiple neurons at a time, and use these experiments to infer probabilistic estimates of the contribution of individual neurons to overall network behaviour.

6.4.1 Timeline

In 1991 Jefferson first evolved synthetic neural networks for a robot control task [219]. The robot agent occupied a 2D grid world, through which it had to navigate along a pre-specified trail. Fitness was assessed by awarding points for visiting landmarks along the trail. The neural network received two inputs specifying whether the cell directly ahead of the robot was on the trail or not. Four outputs allowed the agent to move forward and turn. The network topology was fixed, so only connection weights were evolved.

Later in 1991 Collins evolved neural networks for a virtual life ant colony [73]. He compared several different genotype representations, including ones in which the network topology was fixed to ones where it was evolved. The performance of networks with evolved connectivity was only slightly behind that of networks which had a manually specified task specific topology. This was the first time that the connectivity of a neural network had been evolved for robot control.

In 1994 Gruau used a cellular based developmental encoding to construct neural networks for six legged robot walking [168]. Tree-like developmental programs were evolved to create neural networks from a single precursor cell. The programs obeyed a strict grammar, with each node corresponding to a single developmental instruction, which could instruct the cell to perform parallel and sequential division, change a parameter, or become a neuron. When dividing, two child cells, which inherit all parameters of the parent, are created in either sequence or parallel. Either both will inherit the parent’s connections, or one will inherit the incoming connections, and the other the outgoing. Instruction trees were evolved using the standard “genetic programming” operators of sub-tree crossover and mutation [247]. The neural model was restricted to boolean neurons with integer thresholds and ± 1 weights.

In 1998 Kodjabachian presented the “simple geometry oriented cellular encoding” [237, 238]. This is very similar to Gruau’s cellular encoded, except that development is carried out on a two-dimensional substrate rather than an abstract network. Commands instruct cells to divide, move across the surface, and grow connections in different directions. Like Gruau, Kodjabachian also successfully evolved control for a

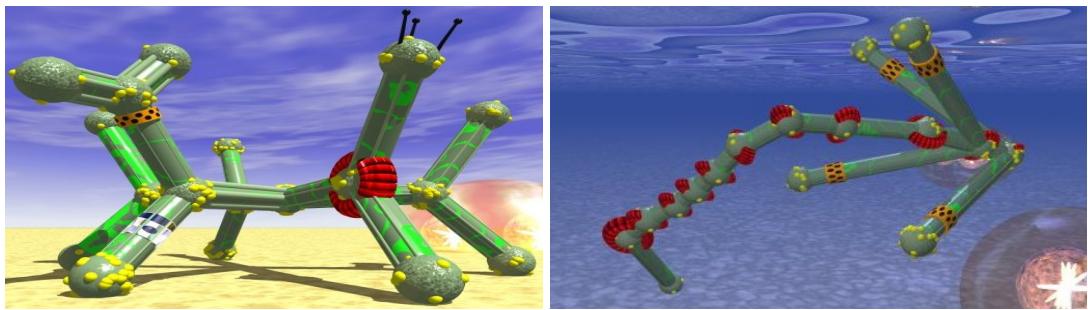


Figure 6.6: *Virtual creatures with fixed morphology and evolved neural control in the Framsticks world.*

Credit for image: Maciej Komosinski [239, 240, 243]

six legged robot.

In 1998 Husbands proposed the “gasnet” model of neural network [210]. The diffusion of nitric oxide molecules had been shown to have a modulatory effect on biological neurons. This challenged the traditional model of a neural network; it was now recognised that neurons could communicate via diffusion of molecules in a 3D space in addition to the direct connections of axons and dendrites. Husbands simulated this diffusion, but simplified to 2D space; each neuron had a 2D coordinate, and generated a signal which diffused in a circle about that point. Control for robot walking was successfully evolved. It was reported that these networks were evolved in 10% of the time required for traditional neural networks.

In 1999 Komosinski released the “Framsticks” evolutionary system, which enables the evolution of neural networks for predefined morphologies (figure 6.6), and the evolution of morphology and control at the same time (see section 6.14).

In 1999 Gallagher used staged evolution to evolve controllers for light following, object discrimination and locomotion in a robot that moves along a one-dimensional line inside a 2D grid world [147]. Pattern generators were evolved first, and then higher level controllers which modulate the activity of the lower levels. The internal structure of modules was maintained by isolating them from the mutation process and only allowing the creation and optimisation of new modules and connections.

In 1999 Reeve investigated the evolution of networks for biped and quadruped walking in a simulated 3D environment (figure 6.7) [349]. A fully connected network, where only connection weights were evolved, was compared to a variable network, where both the existence and weight of connections was evolved. The fitness task was to travel the furthest distance. The fully connected network succeeded in producing

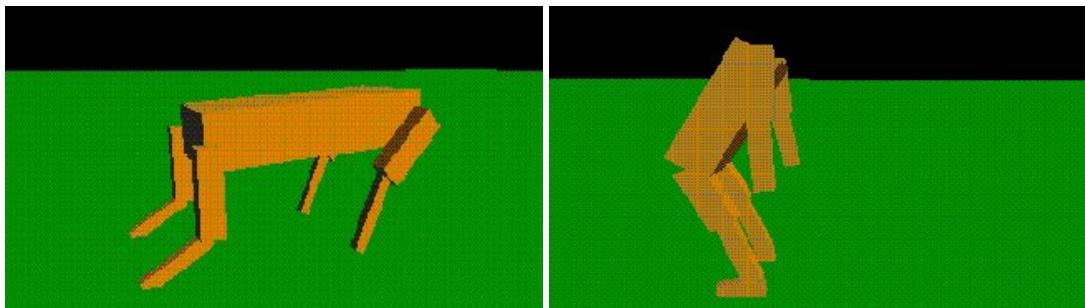


Figure 6.7: *Quadruped and biped walking robots with evolved third order neural controllers.*

Credit for image: Richard Reeve [349]

the greatest rate of motion. The number of neurons necessary for each robot's controller was estimated by analysing the morphological structure of the robot; typical controllers had 50 to 100 neurons. One of the controllers under test exploited morphological symmetry; a single network was evolved and then replicated for each limb, with connectivity between these modules also being evolved. This symmetry had a dramatic effect on performance; all limbs were utilised, and were better coordinated than non-symmetric controllers.

In 2000 Hornby evolved a gait controller for Sony's quadruped AIBO robot [197]. Parameters of a traditional walking controller, such as leg angles, step size, and inter-leg synchronisation timings, were optimised by a genetic algorithm. Fitness evaluation was carried out on a real robot walking over carpets, rubble, and low friction tiles.

In 2001 Floreano compared the performance of evolved “spike response model” (SRM) networks to continuous sigmoid networks controlling a two wheeled Khepera robot [135]. The SRM model uses floating-point arithmetic to calculate the contribution of each spike that has occurred in the last 20ms to the current activation potential of each neuron (see section 3.4.4.2). Although an actual robot was used, control was via a USB connected PC with software simulation of a neural network. The control task was to move the robot around an arena which had walls painted with randomly spaced vertical black and white bars.

The Khepera robot was equipped with a 16-pixel camera, with each sequence of three pixels being filtered with a Laplace transform to extract edge features, and then presented as an input to the neural network. Two other inputs reported the measured speed of each wheel. The neural network consisted of ten neurons, with four of them being used as output signals to drive the motors of each wheel. For details of the

signal coding for inputs and outputs see section 3.3. The fitness function was the sum of wheel velocity over time, which implicitly rewards forward motion and wall avoidance.

Only the connectivity and polarity (inhibition or excitation) of each neuron was evolved. Connection weights and other neuron attributes were fixed and constant. Spiking neural network controllers were successfully evolved to control the robot. However, continuous networks failed to evolve. It was hypothesised that this was due to the restricted parameter set, and that possibly a stateful version of the continuous model (e.g. continuous time recurrent neurons), or the evolution of connection weights, would have been successful.

Analysis of the evolved spiking networks showed that they were well connected, with neurons having a mean of 5 inputs. Redundant neurons and connections were identified, and whilst lesioning some of these neurons caused no significant degradation in fitness, lesioning all of them did, showing that together they made a significant contribution to network behaviour. Many of the neurons were in a constant state of activity, spiking hundreds of times a second, some acting as free oscillators through self-connections. The networks seemed to be using rate coding; two reasons were hypothesised — that the fixed parameters biased the neurons into firing after receiving a single spike, and that the way the input and output neurons signals were interpreted naturally led to a rate coded solution.

In 2002 Reil evolved neural networks for biped walking [355]. The networks consisted of 10 fully connected sigmoid neurons with no sensory input. Connection weights, neuron time constants and biases were evolved. Biped walking is considered a difficult control task, and the evolutionary algorithm had to search a large space since the networks were fully connected. The fitness function measured distance travelled from the starting point, and early termination was performed on solutions that lowered the centre of gravity below some threshold. After successfully evolving biped walking a staged genetic algorithm was used to add a fully connected sensor that enabled the robot to orient itself towards a target point.

In 2002 Frutiger evolved neural control for a swinging monkey [141]. Control was successfully transferred to a physical robot.

In 2002 Stanley proposed the “neuro-evolution of augmenting topologies” (NEAT) algorithm [409, 413]. This algorithm builds topologies of increasing complexity over time, utilising a direct encoding. NEAT solves a few problems with the traditional evolution of neural networks; networks are only complexified if it adds to the fitness,

so there is no unjustified increase in network size, and, unlike most evolutionary neural systems, crossover can be used to combine networks. Every gene is tagged as it is introduced to a network, allowing the phylogenetic lineage of each to be tracked. Crossover can then be carried out on networks which share similar genes by lining up those genes with common ancestry in the same order, and taking excess or disjoint genes from the fitter parent. NEAT also attempts to encourage the evolution of speciation in an attempt to protect diversity and widen the search space.

In 2002 Floreano extended his 2001 research, evolving integer neural networks to control a smaller “Alice” robot, which had infra-red sensors rather than a camera [134]. The vertically striped environment was replaced with a uniformly coloured arena with a cuboid object at its centre, so the robot now had to avoid this object as well as the outer walls. The robot used a PIC microcontroller for both simulating the neural networks and running the genetic algorithm. The neural networks were constrained to only use integer arithmetic so as to be implementable on the 8-bit PIC (see page 64 for details of the neural model).

Later in 2002, Zufferey used the same spiking neural network PIC implementation to evolve controllers for a 3D blimp [495]. Again, evolution was embodied (although by 2005 he had accurately recreated the test arena in computer simulation, figure 6.8). The blimp had sensors for airspeed and range, and three motors to move and rotate.

In 2003 Mahdavi evolved a neural controller for a snake robot using embodied evolution [271]. The snake robot was controlled with muscles made from Nitinol which is a “shape memory alloy”. Shape memory alloys can be trained to store morphology patterns, and will transition between these shapes when a current is applied to them.

In 2004 Tanev evolved neural networks to control a segmented robotic snake inside a 3D simulator (figure 6.9) [430, 431]. Each network was structured as a sequence of “genetic programming” like operations. Two functions were generated, one each for the vertical and horizontal actuators. The same programs were used on every segment of the robot. The evolved controllers demonstrated robustness to physical damage and obstacles.

In 2005 van Breugel evolved controllers for a simulated ornithopter (figure 6.9) and compared evolved sine wave generators with evolved Bézier curve generators [457]. Simple sine wave generators were created for each wing, with the frequency, phase offset, and amplitude being evolved. More complex Bézier waves were then evolved. Bézier control was shown to be more aerodynamically stable, and easier to evolve.

In 2005 Zufferey evolved integer neural network controllers for a flying blimp in-

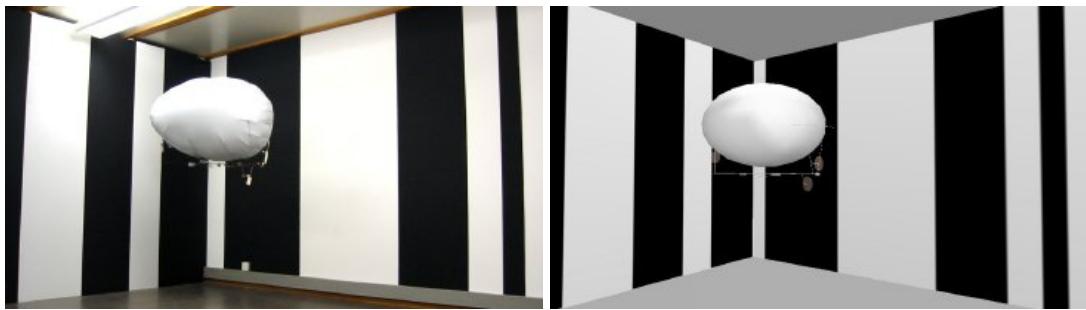


Figure 6.8: *Crossing the reality gap. For this blimp the environment (left) was precisely recreated in simulation (right)*

Credit for image: Jean-Christophe Zufferey [494]

side a simulator and transferred the successful neural networks to reality [493, 494]. This was the same blimp flight task that he had solved with spiking neural networks in 2002. The walls of the environment were patterned with random width vertical stripes. The fitness function penalised collisions with the wall. The simulator recreated the exact environment as well as the blimp dynamics (figure 6.8).

As the blimp was intended to be autonomous the neural network was simulated on a PIC microcontroller using an integer neural network model known as “PIC-NN” (for details see page 73).

In 2006 Der evolved neural control for a spherical robot inside a 3D simulator [104]. The sphere contained gyroscopic sensors which were used as input to a feed-forward network. The output controlled motors which affect the movement of ballast inside the sphere, which in turn generates rolling motion (figure 6.9).

6.5 Evolving reduced continuous neural models

In 1992 Gruau evolved boolean neural networks with binary ± 1 weights and integer thresholds to reproduce 40-input and 50-input logic functions [167]. In 1998 Chiueh evolved discrete ternary networks to carry out classification based on an input/output training set [68]. In 2000 Plagianakos evolved integer neural networks to reproduce 2-input logic functions [337], and in 2006 a controller for a dynamic control task [335].

See section 3.4.6.3 for more information on these models.

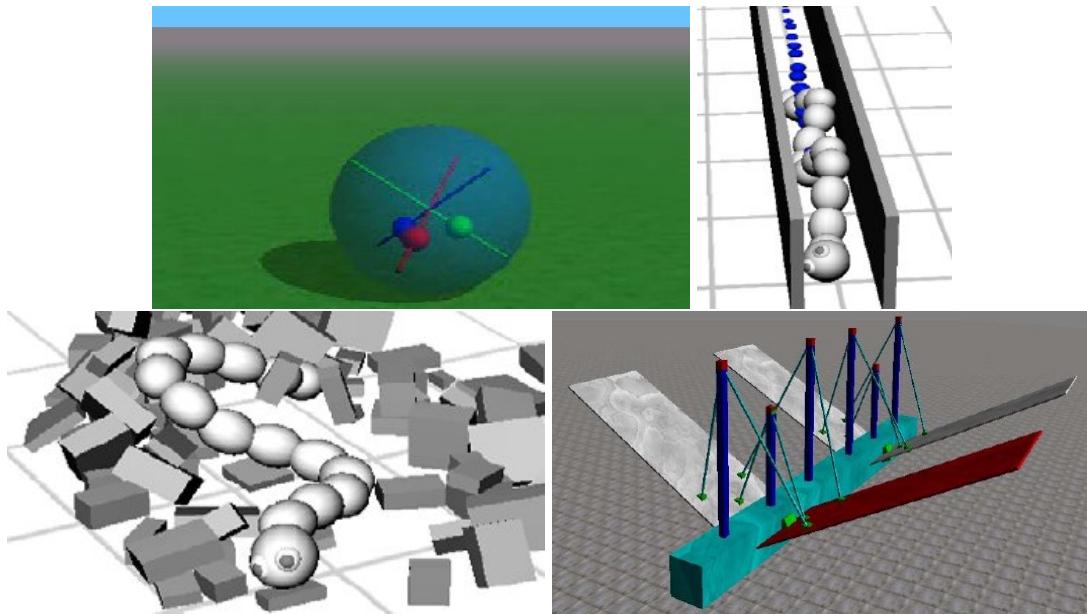


Figure 6.9: Neural networks have been evolved to control sphere, snake, and ornithopter robots.

Credit for image: Ralf Der, Ivan Tanev, Floris van Breugel [104, 430, 431, 457]

6.6 Evolving reduced spiking neural models

In 1999 de Garis evolved cellular automata to reproduce several simple logic functions [91]. Although the systems evolved were cellular automata, a cell being in the 1 state for a single cycle is analogous to a discrete spike occurring over that time period, effectively making this a spiking model. See page 62 for details on the model, and section 6.9.1 for details on the project.

In 2002 Floreano evolved spiking neural networks with integer arithmetic to successfully control a mobile robot [134]. In 2003 Upegui evolved integer arithmetic spiking networks for pattern recognition [455].

See section 3.4.6.1 for more information on these models.

6.7 Evolving modular hierarchical neural networks

It has been argued that the human brain utilises hierarchical networks of modules, each responsible for unique aspects of behavioural and functional ability, and that it is important to emulate this structure in order to create artificial neural networks which control complex behaviour [344]. Modularity may also re-shape the search space to make genetic algorithm based search more efficient. Genomes which allow the expres-

sion of modular repetition can also be smaller.

In 1993 Boers used a developmental Lindenmayer system encoding to evolve modular neural networks [32]. One problem task was discrimination between the characters T and C, each drawn on a 3×3 grid, when translated and rotated inside a larger 4×4 grid. Performance was better than backpropagation on a network with a fixed single hidden layer topology, and the evolved network was smaller. Another problem task was to learn a 10×10 grid in which each cell contains one of four possible values. The two inputs to the network specify the coordinates of a cell, and one of four outputs will become active to indicate which of the four values that cell stores. A topology was evolved which could then be trained with backpropagation. The evolved topology was both smaller and faster to train than a feed-forward with each layer fully connected to the next.

The subsumption architecture and design methodology was proposed by Brooks in 1991 [47]. Although at the time it was not used for neural networks, it subsequently inspired some layered neural architectures [344, 446, 461]. Brooks proposed that robot control could be separated into a hierarchy of modules, where each module would be responsible for performing some distinct action or behaviours. In Brooks's original subsumption architecture each module was implemented with a small finite state machine, having only a handful of states and a few registers to store data. Communication between the modules was asynchronous and unidirectional. Brooks proposed a “bottom-up” methodology for constructing a layered architecture, where lower layers would be implemented and tested first. Once the behaviour of the lower layer was satisfactory, the layer would be fixed and no further changes made to it. Higher level modules would progressively build upon the behaviour of lower layers by inhibiting or overriding specific modules in the lower layer. Brook's subsumption architecture was used successfully to build walking robots that could perform simple tasks like foraging for food, light following, object avoidance etc.. The failure of the architecture to produce robots with more advanced behaviours was attributed to design complexity — the number of unforeseen interactions between different modules increases rapidly as the number of modules is increased — and the lack of a goal conflict resolution or action selection mechanism, which meant that as more behaviours were added, there was increasing conflict between simultaneously active modules.

In 1994 Happel evolved modular networks to discriminate between handwritten digits [174]. The internal connectivity between neurons in the same module was allowed to be dense, whereas connections between modules were sparse. The perfor-

mance of evolved networks with modularity showed a significant, though small, improvement over those without.

In 1999 Prescott described similarities between Brooks's subsumption architecture for robotic control and the layered hierarchies of vertebrate brains and behaviour [344]. He argued that the vertebrate brain had evolved in a similar way to the creation of a subsumption architecture robot, with low-level motor control being used as a base, upon which higher level behaviours were successively built, with conflict resolution between competing behaviours being carried out by a centralised mechanism. According to fossil records, the basic layout of the vertebrate brain has been in place for over 400 million years, and may date back to a mere 50 million years after the Cambrian explosion. Many components of the vertebrate brain have homologous components in non-vertebrates, and the major morphological divisions are found as far back as the earliest fossil records of jawless fish, demonstrating that the basic modularity, functionality, and layout of the brain were discovered very early on in the evolutionary process.

In 2003 Dinerstein showed that the task of evolving neural networks that replicate an unknown non-linear multiple input single output function could be automatically broken down by grouping similar training examples together, evolving smaller networks that learn the function of a single group, and then evolving a multiplexing classifier which selects the correct network output by observation and classifying the input signals [110].

In 2004 Reisinger produced a modular version of NEAT [356]. He co-evolved two populations, one of traditional NEAT networks, and the other of genotypes which combine the NEAT networks into a larger composite network. Performance on the evaluated task (a board game) was shown to be better with the modular approach.

In 2004 Togelius evolved “subsumption architecture” style layered neural networks to control a simulated robot [446]. Togelius implemented the ideas of Brooks's subsumption architecture, but used a neural network to implement each module instead of a finite state machine. Controllers were evolved for light-following and object avoidance, and the evolution of modular controllers was compared to monolithic controllers, showing that the use of modular controllers drastically improving evolvability.

There are other examples of evolved modular neural networks for robot control (section 6.4), and co-evolved with robot morphology (section 6.14). In fact, any developmental genotype coding is likely to encourage modularity.

6.8 Evolving genetic regulatory networks

Genetic algorithms have been successfully used to evolve genetic regulatory networks that match the observations of laboratory experiments; this is done in an attempt to reverse engineer an unknown network after a large amount of experimental data has been collected. Usually it is necessary to evolve both a topology and a model of inhibition/excitation with appropriate weights [5,357,372]. In other cases the connectivity of the genes may be known, but the exact relationship between them (degree of excitation or inhibition) is not [250].

Genetic algorithms have also been used to evolve genetic regulatory networks which utilise variation of gene expression to create visually recognisable patterns, such as the distinctive striping of some insects, or to automate the tedious process of manually writing cell developmental programs [132].

6.9 Evolving cellular automata

Cellular automata are systems with simple state, update rules and interactions only between spatially neighbouring nodes in a fixed topology. Given these constraints, any synchronised complex global behaviour observed must be an emergent property of the local behaviour of individual cells. For a review of evolving cellular automata see [302]

Researchers have designed one-dimensional cellular automata which display emergent behaviour. The firing squad synchronisation problem (see figure 6.10) requires coordinating all of the cells to perform a state change at a single point in time [310]. Many solutions have been proposed, aiming to both achieve the minimal firing time of $2N - 2$, and to minimise the number of states needed [283]. Once the cells have become synchronised they can display arbitrary behaviours, such as simultaneous oscillation, that creates a global pattern. Cellular automata that solve this problem were evolved by Das in 1995 (see section 6.12 and figure 6.14).

Another global challenge is the so called $\frac{1}{p}$ problem (also known as the $p_c = \frac{1}{2}$ task), where all of the cells must change to the state which the majority of them have initially. In a two state 0,1 system, a majority of zeroes in the initial state means the system must converge to every cell being zero, whilst a majority of ones means convergence must be to all ones. To complete this task perfectly requires knowledge of the initial global state; since cells only perceive the state of their immediate neighbours,

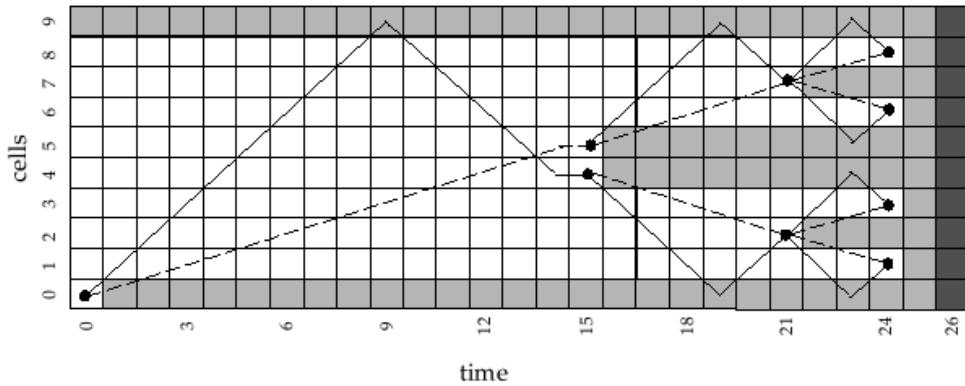


Figure 6.10: *A manually designed solution to the firing squad synchronisation problem. Two signals are propagated at different speeds from the same cell. The faster signal reflects off the remote edge, and the speed ratio is calculated so the signals intersect in the centre. A divide and conquer approach is then used to split the regions until a globally alternating, locally recognisable, pattern is reached, thus synchronising all cells.*

Credit for image: Melanie Mitchell and P. C. Fischer [302]

and perfect knowledge of the initial state is destroyed as soon as the first transitions occur, there can be no perfect solution that works under all initial conditions.

In 1993, Mitchell successfully used genetic algorithms to evolve cellular automata rules that can be used on this problem (figure 6.11) [79, 305]. The cellular automata contained 149 cells, producing a search space with size 2^{149} . This was too large to exhaustively evaluate with a fitness function, so instead the fitness function only sampled 100 possible initial conditions, which were randomly chosen for each generation. The 100 possible initial conditions were biased to have a majority of either 0s or 1s by drawing samples from a distribution where p was the fixed probability of a cell being 0 or 1, and $p \in [0, 1]$ was uniformly distributed, rather than having samples being randomly drawn from all of the possible configurations in the 2^{149} space. The reason given for this was that a random selection of 149 initial states with $p = \frac{1}{2}$ for each would form a binomial distribution with all configurations having $p \sim \frac{1}{2}$, biasing the test cases around a difficult area of the search space to classify — in fact, due to the central limit theorem and n and p being relatively large, the resulting distribution will be approximately normal: $N(np, np(1 - p)) = N(74.5, 37.25)$ so about 95% of the test cases would have the number of cells initially set to 1 lying in $[68.4, 80.6]$. This meant that it was possible for a solution to get a perfect fitness score by successfully classi-

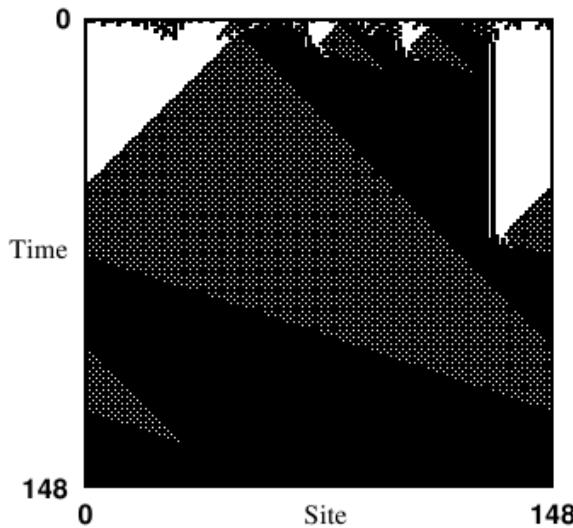


Figure 6.11: A cellular automaton evolved to solve the $\frac{1}{p}$ classification task, in which the final state of all cells should be the one in which the majority of them are initially. Credit for image: Crutchfield and Mitchell [78]

fying the small, biased sample — it is reported that the evolved rules did manage this stage after only 20 or so generations. No details were given of how well the best rules would perform on the general population of random initial conditions.

In 1997 Sipper evolved synchronous non-uniform cellular automata (NUCA) to solve the $\frac{1}{p}$ task, and the synchronisation task [401]. He also devised an “average cellular distance” metric to quantify connectivity, and showed that this linearly correlated with performance. This was explained by the hypothesis that on global tasks cellular automata will perform better when information can travel faster between nodes. It was then shown that high performance and low connectivity architectures could be evolved, and that populations would cluster around points of low average cellular distance, thus taking advantage of better connectivity without being forced to explicitly specify network topology.

In 1998 Sipper evolved “globally asynchronous locally synchronous” NUCAs to solve the density and synchronisation tasks , and found their performance comparable to synchronous cellular automata [402]. Due to their non-deterministic nature an exact solution to the synchronisation task is not possible, but with certain updating schemes a logically equivalent progression can be defined.

Cellular automata have been evolved for a variety of pattern generation tasks, see section 6.12.

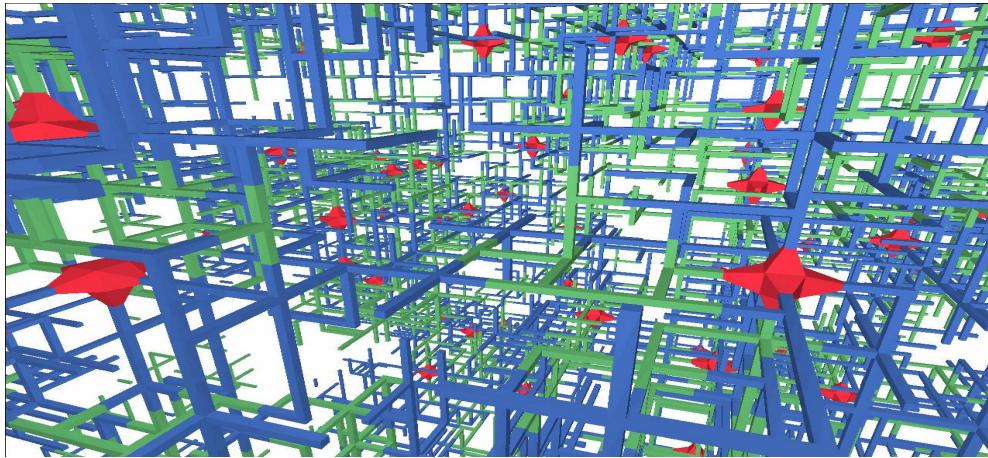


Figure 6.12: *The CAM-Brain project aimed to create functioning brains by constructing neural pathways from 3D cellular automata.*

Credit for image: Genobyte, Inc.

6.9.1 Evolving cellular automata neural networks

In 1993 de Garis, working at Japan’s “Advanced Telecommunications Research Institute” (ATR), started the “cellular automata machine brain” (CAM-Brain) project [88]. He aimed to carry out artificial evolution of cellular automata, the cells of which would implement the functionality of a neural network, and be implemented using networks of programmable hardware (image 6.12) [87, 91]. He claimed that this would allow the evolution of artificially intelligent systems to occur “at electronic speeds”, because electronic circuits can switch faster than biological neurons, and the entire genetic algorithm process would be carried out intrinsically in hardware without any need for slow external inputs.

By 1994 a two-dimensional cellular automata simulator had been developed, and a genetic algorithm was used to evolve integer neural networks that output a constant value (point attractor), and that produced an oscillating output (cyclic attractor). By 1996 a simulator of 3D cellular automata was developed. The creation of these simulators took a long time as, according to de Garis, it was necessary to hand-craft 11,000 rules for the 2D version, and 150,000 rules for the 3D one. Precisely why so many rules were needed, why this could not be automated, and how they were checked for correctness, was not discussed. A binary state 2D CA, where each cell communicates with four neighbours, requires only 32 rules. These cellular automata, however, required many states as they had to contain integer neural networks, with a unique cell state for each discrete integer. It does not seem as though this should have caused any

greater difficulty though, since these extra states only affect functioning of the neuron, which uses a standard sum-and-threshold model which should not require the manual definition of thousands of rules.

The fact that the system was 3D and stored orientations for each cell also should not have added substantially to the complexity, as the cell rules would be symmetrical about the six possible orientations. In later papers de Garis hints that these early systems were not “gridded”, by which he means the developmental substrate space was continuous and non-synchronous, so signals could arise out of phase, but this is odd, as a regular geometry and synchronous timing are two of the defining features of cellular automata.

Despite a decade of research, over \$1.4 million of grant funding, and the lofty claims of creating an artificial brain by 2001, this project failed to produce any working AI system. No papers were published reporting the successful evolution of any intelligent control systems, and none of the current owners of the hardware developed, which include ATR, the creditors of both Starlab and “Lernout and Hauspie” (which both went bankrupt in 2001), and the designer of the CBM itself, Michael Korkin (former owner of the now defunct company Genobyte Inc.), have published any information regarding successful or working intelligent systems [92].

Nevertheless, the basic concept of utilising genetic algorithms to evolve cellular automata with custom hardware is an interesting one. The main problem faced is that of choosing a workable fitness function, and performing the fitness evaluations. It was claimed that complete brain systems could be evolved in seconds, with billions of fitness evaluations occurring, and yet it is clear that any evaluation of fitness on a real world task is going to require either a robotic body, or a simulator, both of which will take a significant time to run a single fitness evaluation. de Garis suggested that an on-chip intrinsic evaluation function that computed the sum of squares error between the output of the cellular automata and a reference waveform would be sufficient [91]. Whilst it may be possible to replicate simple non-linear analog functions this way, it implies that the function is already known, or that there is a way to generate the output of the function within the same timescale of the evolving FPGA. This is unlikely to be the case for most tasks of interest, particularly the simulation of artificial brains to control complex robotic systems.

Initially experiments were carried out using either simulation software running on a standard workstation, or a much simplified model running on MIT’s CAM-8 “cellular automata machine” [445]. The CAM-8 was a PCI card developed in 1989 which could

quickly process cellular automata held in 16-bit memory. Each 16-bit word could be divided into sections which would be independently shifted to construct a new address used to access the memory on the next cycle. For a 2D CA with a neighbourhood of 5 cells this meant each cell could have a 3-bit state. However, at this stage de Garis said that it was not possible to implement CAM-Brain with only 3 bits per cell [89].

In 1996 it was realised that the complex integer networks initially envisaged, devised from hand-crafted cellular automata rules, were too complicated to be implemented in hardware. A new “collect and distribute” (CoDi) neural network model was presented, which used a 4-bit accumulator to sum inputs, and then fired if the result exceeded some constant threshold [151]. They decided to build a custom cellular automata machine using FPGAs, and technology constraints led to the neural model being further simplified with 1-bit signalling, resulting in the “CoDi-1Bit” model [152].

The new design was now a binary state cellular automaton, with 6 neighbours per cell. The initial development phase had been modified so that each cell now had an orientation; previously, the cellular automata had been like a blank canvas, with all growth instructions being sent out along developmental axons and dendrites. The new model could be accurately described as a spiking neural network with uniform 3D topology, as it was built from digital logic adders, threshold comparators, etc., and not automata cells. It was shown, in software simulation, that the system could evolve cellular automata that generate an oscillating output bit, change the distribution of output ones and zeroes in response to an input bit being flipped, and discriminate and classify simple input patterns [91, 152].

The actual hardware implementation, termed “CAM-Brain Machine” (CBM), was to be built by one of the researchers involved in the project, Michael Korkin, who established the US company Genobyte to market the machines commercially [93]. It would potentially be capable of simulating 37 million neurons, each of which would be created from hundreds to thousands of cellular automata cells stored in a 1.2 GB distributed memory, with each neuron being updated hundreds of times a second. This would be an estimated 750 times faster than the CAM-8.

Like the CAM-8, it was implemented as a PCI card, so that it could be accessed over a fast bus by the host PC. It would use 72 Xilinx XC6264 FPGAs, each on a daughter card with 16MB of local memory, connected via a backplane, to implement a single $24 \times 24 \times 24$ cell module in hardware. A robot control system would consist of many of these modules, connected via some network, with each individual module having up to 180 inputs and 3 outputs. For each output a 96-bit spike train would be

recorded in a central shared memory, so that other modules could use it as an input. The system would use time-sharing to carry out the processing of each module; each FPGA would swap-in cell states from the local memory, together they would simulate a cube for 96 cycles, and then write back the cell states to local memory, and the 96 bits generated by each output to the shared memory.

Once it was accepted that a single-bit binary protocol was necessary due to hardware requirements, further research work focused on signal coding [87, 244]. The traditional interpretation of biological spiking is to interpret the frequency of the spiking as the value that is being conveyed; so called rate coding. This was rejected as being “too slow”, as it requires a short time period in which to count the spikes before a value is available. Unary coding, in which the coded value is simply the number of cells from some group that are simultaneously active, was rejected as being “too jerky”, since many bits may switch at the same time producing large discontinuities. It is possible that other codings of the multiple bits, such as a “one-hot” or “m-of-n scheme”, would not exhibit this problem, but this was not considered.

Ultimately the “spike interval information coding” (SIIC) was chosen, which uses a convolution filter over binary bitstreams to generate a continuous floating-point output. The “Hough spiker algorithm” (HSA) performs the inverse function of converting integers into a binary bitstream. For more details of these functions see page 47. To demonstrate that a genetic algorithm could evolve cellular automata that successfully utilise this coding a “CoDi” module was evolved which generated an oscillating SIIC coded output that roughly resembled a sine wave, and another module was evolved that phase shifted its SIIC encoded input signal by $\frac{\pi}{4}$ [316].

In 1998 the contradiction of being able to compute appropriate fitness functions that were both realistic and intrinsic was recognised [87]. The use of a physics simulator was described as a “necessary evil”, but it was stated that only the low-level modules directly involved in motor control would need to be evolved in simulation. Higher levels, which would greatly outnumber the lower levels, would just reproduce activation patterns and behaviours specified by a human designer, and hence could be evolved intrinsically. This does not seem practical, as the human designer would need to anticipate all of the potential conflicts and interactions between the many active components, thus removing the advantages of using a genetic algorithm in the first place. It should also be noted that the generation of low-level motor patterns is not a problem for researchers — it is precisely the issue of combining them together into coherent behaviours, arbitrated and moderated by some task based planning system

with its own world model, that has been the dominant problem for AI research.

Some of the published papers may give the impression that the FPGA based “CAM-Brain Machine” worked, and had been successfully used to evolve functional neural networks. However, other papers state that the research was done through simulation. On his web page de Garis states that the CBM was only functional for about one month [90]. The scheduled delivery of a CBM to ATR had been delayed several years, and arrived only a few months before de Garis moved to Starlab. A CBM was delivered to Starlab in summer 2000, but hardware problems (attributable to either the CBM itself, or to inadequate cooling and an unstable power supply) had prevented it from working correctly, and then Korkin, who had designed the hardware and was working on the firmware, remotely disabled the CBM in an attempt to extract payment for money he was owed. Starlab, the Belgian research organisation which was now employing them both, went bankrupt shortly later, and thus a fully working “CAM-Brain Machine” was never completed [130].

6.10 Evolving analog circuits

Genetic algorithms have been used to successfully evolve analog circuits, both in simulation, and on FPGAs. In 1995 Thompson evolved a (partly digital) robot controller [441]. In 1996 he evolved digital filters and frequency discriminators [440, 442, 443]. In 1997 Koza evolved analog circuits using genetic programming [248]. In 1999 Mazumder used genetic algorithms to optimise the layout of analog and digital circuits [284]. Gallagher evolved analog circuits that implemented pulse-coded continuous time recurrent neural networks to control locomotion in a six legged robot [146].

6.10.1 Evolving FPAA robot control

In 2005 Berenson used a “field programmable analog array” (FPAA) to evolve artificial neural networks to control biped walking and fault recovery in a real robot created by 3D thermoplastic printing (for details of this process see page 203) [24]. Hardware resources were severely constrained, with networks having a maximum of four neurons, each having three inputs. The quadruped model was the same used in Bongard’s research on evolving neural control and morphology [35].

6.10.2 Evolving FPGA oscillators

Boolean oscillators generate a single binary output which toggles between 0 and 1 at some desired frequency. They are commonly used in electronics to produce clock signals which synchronise the flow of data through circuits. The usual way of doing this is to utilise a crystal oscillator, which is a circuit based around a piece of crystal which has a mechanical resonance at which it will vibrate when a voltage is passed through it, thus producing a varying output voltage at some specific frequency which is further processed through amplification and filtering to produce a clean and stable square wave.

It has long been known that it is possible to create free running clock signals by constructing an inverter chain, i.e. a loop of gates which contain an odd number of NOT operators so that the loop can not become stable and values will be propagated around it forever. Thompson showed that it was possible to intrinsically evolve such a system inside an FPGA using genetic algorithms [443]. The fitness function was a measurement of how closely the signal on some output pin of the FPGA matched the desired frequency of oscillation.

6.10.3 Evolving FPGA frequency discriminators

Thompson evolved circuits inside a digital FPGA [440]. Despite the FPGA being designed for digital, synchronous use, the representation of the FPGA's internal circuits and the format of its programming bitstream do not enforce these restrictions. This allowed the genetic algorithm to explore a wider solution space that potentially included asynchronous and analog effects. The FPGA programming bitstream was 1800 bits long; too great for an exhaustive search.

The input to the FPGA was a single bit that varied between two regular frequencies. The output was a single bit that was supposed to discriminate and classify the input signal into the two known frequency bands. In order to encourage the evolution process towards a novel design no clock signal was provided to the FPGA. This meant the FPGA would have to somehow derive an internal model of timing with which it could reference the input signal. It also meant that the evolved design would be completely asynchronous.

It was found that a solution to this problem could be successfully evolved on the FPGA. Analysis of the evolved programming bitstream showed that it worked in a unexpected way. Thompson hypothesised that it was deriving timing information from

an unexpected source such as the power supply (UK mains is 50Hz), either through some internal mechanism of the FPGA, or through signal crosstalk. The analog non-linear properties of the solution were evident when it was shown that the discrimination frequency varied with temperature.

Strangely, it was found that the bitstream contained programming that was redundant and could theoretically be removed, and yet when this was attempted experimentally, the “clean” programming failed the frequency discrimination task. Thus, the genetic algorithm had discovered a working solution very different from that which a human designer would have created. The evolved design was closely coupled to the underlying hardware, and utilised properties unique to the particular FPGA upon which it was evaluated, with its performance degrading when run on another, supposedly identical, FPGA of the same family.

Later analysis showed that a typical evolved frequency discriminator circuit derives timing information from groups of gates with recurrent connections, similar to the inverter chain used for delay matching in asynchronous circuits (figure 6.13) [444]. It is interesting to note that analysis of evolved asynchronous random boolean networks also revealed that inverter chains were the principal method of timing when generating rhythmic patterns of activity [364].

6.10.4 Evolving on an analog FPTA

Keymeulen and his fellow researchers built a hardware evolutionary system based on a custom “field programmable transistor array” (FPTA) fabricated in 50nm technology [230]. The system was designed to evolve analog transistor networks with the aim of replicating analog functions. The fitness evaluations were performed on-chip.

The performance of the system was compared to the same fitness evaluations using a SPICE [346] software model of the chip, running on a 256 CPU supercomputer. It was discovered that the performance of the on-chip fitness evaluation was equivalent to 128 of the supercomputer processors working in parallel running SPICE software simulations. The system was used to successfully evolve both a Gaussian filter and AM bandpass filter in less than 4 minutes; evolution in software on a single CPU took over 4 hours. In general, it was found that on-chip fitness evaluations were 20 to 50 times faster than their software equivalents running on a single CPU.

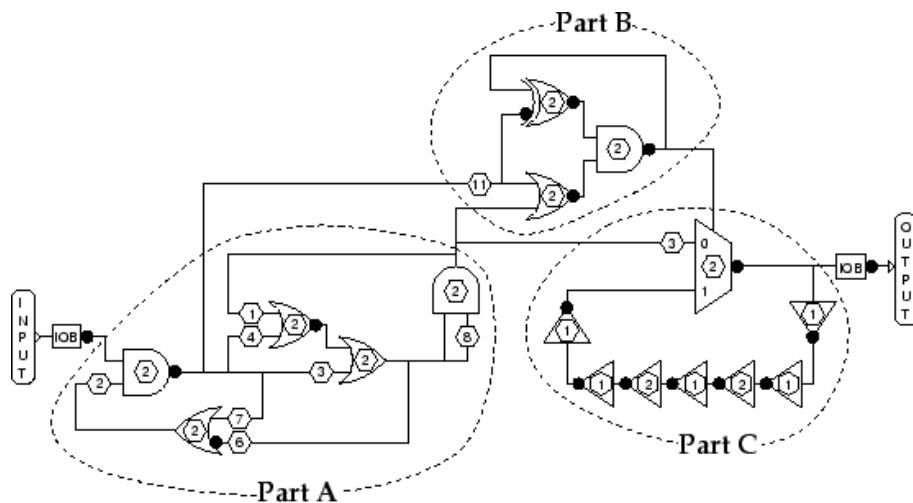


Figure 6.13: One of Thompson's evolved frequency discriminators. Despite the circuit being digital, and being implemented on a digital FPGA, its behaviour is asynchronous and utilises non-linear analog effects. Performance was found to degrade when this circuit was implemented on a supposedly identical FPGA from the same family as that used for evolutionary fitness evaluations, showing that the evolved circuit was intrinsically linked to the underlying hardware.

Credit for image: Adrian Thompson [444]

6.10.5 Evolving fault tolerance

Fault tolerance is a desirable property for circuits to have. As designs move towards smaller feature sizes and larger gate counts, the probability of manufacturing errors increases. One approach being employed by traditional designers is to incorporate redundant circuits which can be disabled if they fail post-manufacture testing. The success of this approach relies on the yield tending towards producing single module failure, as the recovery from failure of more components than were designated as redundant is not possible. It also is not possible to recover from failures in modules which are not redundant. Hence the human designer has to anticipate which modules are suitable for redundancy; the ones which tend to be used in highly parallel, repetitive arrangements, such as the arrays of a DRAM or FLASH memory array, or the processing elements (PEs) of IBM's Cell processor.

There has been much work done on using genetic algorithms to evolve and optimise fault tolerant circuits. Hartmann showed that two bit adder and multiplier circuits could be evolved to be resistant to signal fluctuations [177]. Although the evolved circuit designs were digital, the

simulation environment modelled analog components.

6.11 Evolving digital circuits

It is difficult to create evolved versions of traditional synchronous digital circuit designs. This is due to the fact that the desired functionality of digital circuits tends to be very precise; it is difficult to see how multi-stage stateful logic circuits, such as pipelined CPUs or encryption engines, could evolve from simpler basic blocks. Therefore most of the research activity has focused on evolving circuits that act asynchronously. Despite this, these circuits are quite different from those which would be manually designed by the asynchronous circuit research community using automated state machine synthesis tools and C-element and arbiter design primitives.

Evolving digital circuits is appealing since simulation of digital circuits on generic processors is fast, and they can be easily transplanted into hardware FPGAs. In contrast, simulation of analog circuits is very slow, and hardware instantiation requires the design of full-custom “application specific integrated circuits” (ASICs), which is a slow and expensive process.

Evolved digital circuits have not managed to reproduce the complexity of manually created designs. In 2002 the most complex circuits evolved were simple binary adders [164]. The most complex design evolved since then was in 2005, when Stomeo evolved 4-bit multipliers, with 8 inputs and 8 outputs, typically containing less than 30 logic gates [418]. In contrast, a 2003 Intel Pentium-M CPU had around 77 million gates, and a 2006 Xilinx Virtex-4 FPGA supported up to 24 million programmable gates.

There is a body of work that crosses over between the evolution of analog and digital circuits. In particular, Thompson’s work on evolving circuits within FPGAs is difficult to classify [440, 442, 443]. FPGAs are traditionally used to implement synchronous, digital circuits, and the evolved circuits obviously rely heavily on the digital look up tables contained within the FPGAs. However, further analysis of the evolved circuits shows dependencies on analog and non-linear behaviour, and as no external clock signal was provided the evolved circuits are best classified as asynchronous analog, and are therefore covered in section 6.10. Thomson focuses on “intrinsic evolution”, in which the evolutionary process is carried out on the actual target FPGA device, as opposed to some family of devices, or a logic simulator.

There has been almost no published work on the use of genetic algorithms to evolve

the kind of digital circuits that would be created by asynchronous circuit designers, that is, those that use hazard free logic, C-elements and arbiters to manage synchronisation. This is possibly due to a lack of envisaged utility, lack of interest, the niche nature of the intersecting research areas, or to the difficulty of the task (asynchronous circuits can easily deadlock) leading to experimental failure which has gone unreported. In 2005 Shanthi used a two stage evolutionary process to evolve a modulo-6 counter and a 4-state benchmark circuit with minimal hazards [389].

6.11.1 Optimising gate count

Vassilev described a genetic algorithm that could be used to create and optimise combinational digital circuits [460]. By defining “neutral networks” as flat areas of the fitness landscape, only allowing evolution to proceed around these areas, and starting with fully working manually designed solutions, the genetic algorithm was constrained to only produce correct results.

The fitness function penalised large circuits, so the genetic algorithm would tend towards minimising the circuit size. A cellular substrate was defined for the actual evolution, which allowed potential subcircuits to evolve alongside the main circuit, with mutation allowing these subcircuits to replace sections of the genome. This had the desired effect of always maintaining or decreasing the size of the functionally necessary parts of the circuit, whilst giving the genetic algorithm room to experiment with new subcircuits. It had previously been shown that having such a “scratch pad” space was necessary for successful evolution in this domain.

The genetic algorithm reduced the gate count of the manually designed 3-bit multiplier by 23.3%, and the 4-bit multiplier by 10.9%. Unfortunately, there is a large cost associated with verifying the correctness of the mutated solutions — every possible input pattern must be presented to the new circuit, and the output computed and compared to the fully functional reference circuit. This is practical for small circuits with a low number of inputs, but quickly becomes impractical for larger circuits.

The other problem with this type of optimisation is that it is only possible for feed-forward combinational circuits; optimising a circuit that contains state, such as feedback loops, or components such as flip-flops and C-elements, across the state boundaries, is not possible. It would, however, be possible to automatically extract the combinational circuit elements from a larger design, individually optimise them, and then replace the original subcircuits with the new optimised designs.

6.11.2 Optimising power

Several researchers have experimented with the use of genetic algorithms for producing lower power circuits [7, 58, 59, 76, 464]. This is generally useful for mobile devices, but also has applications in test control.

6.11.2.1 Optimising power for normal operation

There are various techniques for lowering the power consumption of a circuit. One approach is to increase the efficiency of the circuit by optimising the original design in order to shorten the critical path. In general, a shorter critical path enables a shorter clock period to be used, which increases a circuit's speed, and hence increases its power consumption. However, when optimising for low power, it is better to decrease the operating frequency, which reduces transistor switching and lowers power consumption. Lowering the supply voltage decreases the current drawn when transistors switch, making their transition slower. The net effect of this is that by optimising the critical path of a circuit, the potential clock frequency will increase, and by reducing the supply voltage, the potential clock frequency will decrease. These two effects can be made to cancel each other out, in order to maintain the same performance as the previously unoptimised design, but with lower power consumption.

Typical optimisations to reduce the length of the critical path are to reduce the size of state machines and carry out new pipelining, timing, and synthesis procedures. This is the approach taken by Arslan, who used genetic algorithms to optimise a digital signal processor [7]. Test results showed a greater than 50% power saving whilst running a common DSP filter with evolved optimisations.

A more unconventional approach was taken by Venkataraman [464]. State machine partitioning is a mechanism for splitting a state machine into separate state machines which collectively implement the original desired behaviour. The aim of this is to allow the circuits controlled by individual state machines (and indeed, the state machines themselves) to be turned off, hence saving power. This will not work for every circuit — the original state machine must be amenable to partitioning. Typically this means that the states can be divided into two or more sets, where transitions normally occur within these sets, and rarely between them. In this research the genetic algorithm was applied to discover good partitions.

The genetic algorithm based approach was able to reduce power consumption over standard synthesis by an average of 57% at the cost of increasing the area by 77%.

It also compared favourably to a conventional synthesis tool targeting low power; on this test set the genetic algorithm approach was found to reduce power consumption by 36%, while the conventional tool only managed an average of 16%.

6.11.2.2 Optimising power for test control

Modern processors and other complex ASICs are designed with complex “built in self test” (BIST) circuits that are used after manufacturing to verify that the circuit works. In the case of processors, this testing is used to grade the quality of the chip by finding the highest clock frequency or lowest voltage that can be presented to it without causing internal operating errors to occur. This grading is then used to differentiate pricing in the market to increase the value of low voltage (i.e. low power) and high speed parts. For most devices, post-manufacture testing is used to simply establish that the device works correctly at the limits of its approved operating environment.

Testing is performed by downloading test bitstreams to the device under test via its “Joint Test Action Group” (JTAG) port. The test bitstreams define activation patterns for the self-test circuits, and alter the logical structure of the chip, e.g. by tri-stating bus buffers. The patterns are typically auto-generated by some high-level electronic design tool, and are designed to stress test all the circuits of the chip in parallel. These tests produce an artificially high level of activity within the chip, leading to an excess of heat being generated. This heat dissipation can actually be a problem in chip testing, as portable chips are typically designed to conserve power, and have lower thermal design ratings and poor heat dissipation.

Genetic algorithms were used to evolve test patterns which minimise energy consumption [58, 76]. The test patterns were modified by introducing redundant bits, and then broken down into sequences which test for different faults. The problem was then to choose which sequences to gather together into sets in order to cover the same number of faults as the original test pattern, whilst minimising the power requirements. The addition of redundant bits is performed using a genetic algorithm based tool. The fitness of sequences is assessed by running a power estimator tool, and the selection of sets of sequences is then carried out by another genetic algorithm.

It was found that power consumption of the evolved designs was 45% to 85% less than the original test patterns, with the same fault coverage.

6.11.3 Evolving digital circuits for robot control

Evolved robot control systems tend to utilise analog or real-valued components. However, there has been some work done on the evolution of complete digital controllers.

Thompson evolved a robot controller to perform the familiar wall avoidance task [441]. The robot controller consisted of sonar sensors, a DRAM array, some asynchronous registers, and outputs to a pulse-width modulation motor driver. The DRAM array was used as a lookup table, with the input address being the sonar sensor values and some bits fed back from the DRAM output.

The hardware design appears to have been deliberately chosen as one which would be rejected by a human designer. The unpredictable nature of the asynchronous registers, the use of a clock signal selected by the genetic algorithm, and the rapid cycling through states that would occur as the system “jumps” through different memory instructions due to input signal changes, all conspire to make the task of manually defining a bitstream unlikely to succeed.

The lookup table contents were then subjected to the evolutionary pressures of the genetic algorithm. Within a few thousand generations the genetic algorithm found solutions that could correctly solve the task and navigate to the centre of the arena, even when initially placed facing a corner. The transition tables of the successfully evolved circuits were found to be complex and closely coupled to the actual hardware (e.g. the motor for one wheel was physically faster than the other, and the circuit had correctly compensated for this).

6.12 Evolving pattern generators

Nature surrounds us with recognisable patterns that occur both geometrically and temporally. As we have little understanding of how to orchestrate small scale localised interactions into producing the kind of emergent global behaviour that describes these patterns, researchers have utilised genetic algorithms to create various types of networks which converge to produce either single patterned states, or cyclic patterns of activity. The creation of globally recognisable patterns from the localised interactions of large numbers of simple computational devices is an instance of emergent behaviour, and we may be able to learn how to build similar systems by analysing the evolved rules. The generation of non-uniform cellular automata and boolean networks that converge from a random state towards specific patterns is known to be an NP complete

problem [232].

Generation of cyclic patterns underlies low-level biological motor control (see section 2.4). As section 3.6 points out, for many tasks cyclic activity can be driven by changing input patterns, rather than by the purely internal activity of a pattern generator. In fact, it has been shown that enforced patterns of input activity can have a severe impact on the global activity of what would otherwise be a closed system; studies on asynchronous cellular automata have shown that continuous perturbations, apart from obviously preventing the system from being in a single point attractor state, make these cellular automata develop large scale regular spatial structures, with long range correlation between cell states, and that these structures are stable despite the continuous perturbations caused by the environment [489].

6.12.1 Timeline

In 1995 Das evolved solutions to the cellular automata synchronisation problem, in which all the cells must simultaneously perform a state transition, and, in this case, oscillate afterwards to create a cyclic pattern (figure 6.14) [83]. Analysis of the evolved solutions showed that they rely on the boundaries of areas of uniform state colliding and interacting to perform computation, resulting in the destruction and formation of these areas. These boundaries can be viewed as “particles”, travelling across a two-dimensional time and space substrate, which collide and interact with each other to form a system of soliton computation.

In 1997 Harvey showed that asynchronous random boolean networks possess point attractors, but far fewer than their synchronous counterparts [179]. This showed that it was possible for the distributed unsynchronised updates of an asynchronous boolean network to produce the same kind of stable attractor dynamics that were known to be possible with synchronous or continuous networks. Attractors were far rarer though; the expected number in any network is 1, and the distribution is heavily skewed towards a small number of attractors, although for $2 < K < 3$, corresponding to Kauffman’s “edge of chaos” transition, there are a small number of networks which possess a large number of attractors.

In 1997 Sipper evolved semi-asynchronous non-uniform cellular automata to solve the synchronisation and density classification tasks [401, 402]. Due to their non-deterministic nature, asynchronous cellular automata can not reproduce exact patterns. Sipper used cellular automata that were synchronous within small neighbourhoods,

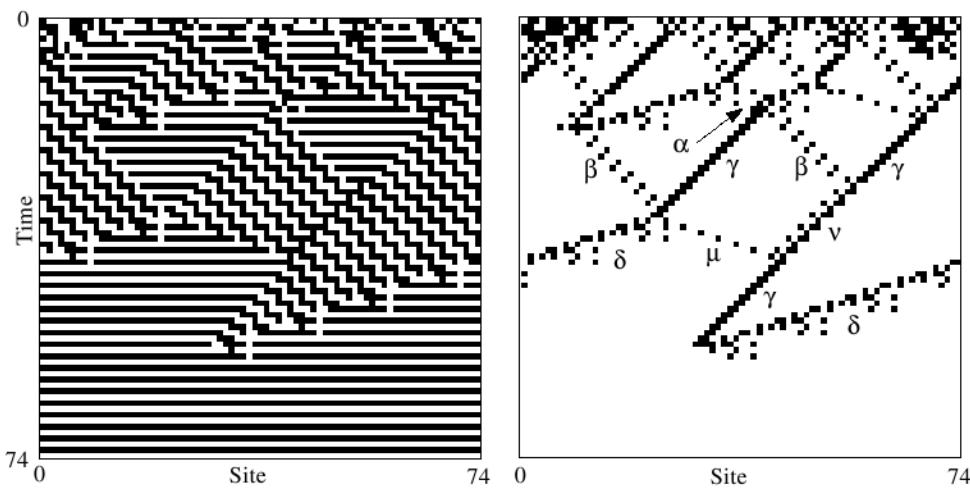


Figure 6.14: *Space-time diagram of a CA evolved to solve the synchronisation task. The initial state is random. In the left image it appears that there are triangular shaped blocks containing the same pattern overlaid on top of one another. The boundaries of these blocks can be viewed as “particles” which move across the CA in a filtered diagram (right), with collisions either wiping out the colliding particles, or performing computation producing new particles.*

Credit for image: Das, Crutchfield, Mitchell and Hanson [83]

but globally asynchronous (this would be known as “globally asynchronous locally synchronous” in asynchronous circuit terminology).

In 2000 Di Paolo defined rhythmic attractors as being similar, but not necessarily identical, repetitive cycles of activity, and used a genetic algorithm to evolve asynchronous random boolean networks displaying this behaviour [107, 108]. The fitness function initialised networks into a random state, and then observed each state as transition rules were applied, scoring networks highly if they appeared to be looping through similar states with a long limit cycle. Rhythmic networks were successfully evolved (see figure 6.15).

A later analysis of the networks showed that most evolved rhythmic activity relied on rings of signal propagating cells, similar to the well known inverter chains from asynchronous circuit design (figure 6.16). These networks were not resilient; disruption to the chains severely impacted the network’s ability to generate rhythmic behaviour [364]. Biological neural networks display a high level of resilience, which suggests that the evolved networks are not biologically plausible, however, resilience to damage was not assessed by the fitness function. If it was, networks may have been encouraged to develop robustness.



Figure 6.15: An evolved asynchronous random boolean network that displays rhythmic behaviour. This network has 32 nodes, each with 3 inputs. The network state is plotted as a series of vertical lines, with time progressing from left to right over 1000 cell updates. Although the network is oscillating between two states, at any particular time (i.e. if a vertical line were to be drawn at any point) the network would be in a mixed state. Note that the repeating patterns are clearly not identical.

Credit for image: Ezequiel A. Di Paolo [107]

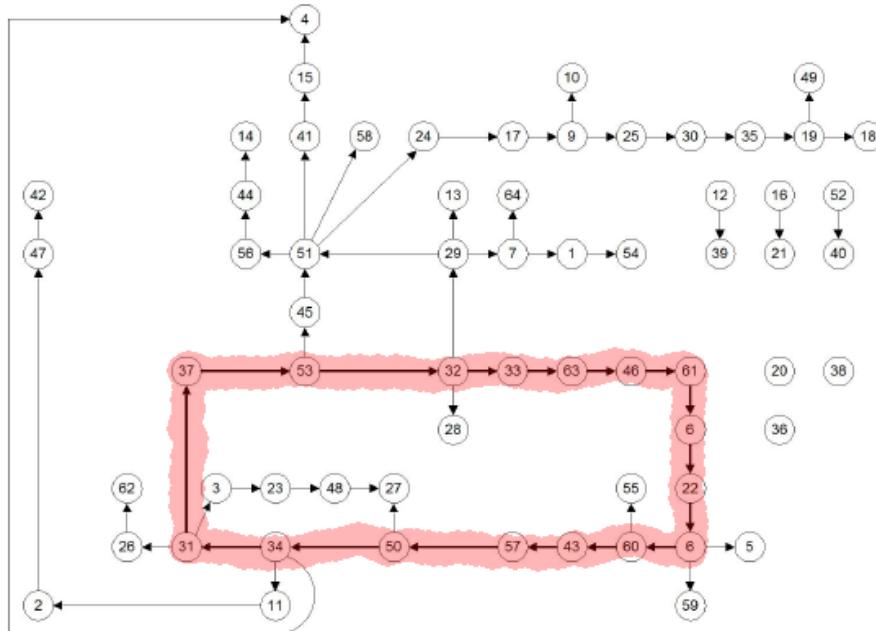


Figure 6.16: This is a plot of a random boolean network that was evolved to exhibit rhythmic behaviour. Analysis of such networks showed that most relied on evolved cyclic structures which propagated and inverted a signal in order to produce timing information. Here this cyclic timing structure has been highlighted; it can be seen that it constitutes rather a large part of the whole circuit. This method of generating a timing signal is similar to the inverter chains, or free running clocks, of asynchronous circuits.

Credit for image: Ezequiel A. Di Paolo [364]

In 2003 Basanta used genetic algorithms to evolve cellular automata that develop globally recognisable crystalline microstructure patterns when run [15]. Differing microstructural arrangements can change the way that materials behave, even though they are composed of the same molecules. For example, metallurgic heat treatment is used to strengthen the alloys used to manufacture swords and jet engines, to make them stronger and increase their resistance to high temperatures.

In the metals used to build jet engines one of the primary factors that determines strength and heat resistance is the dispersion and size of small spherical alumina crystals embedded in larger nickel-aluminium crystals. This can be modelled using a cellular automaton, and physical properties can be evaluated using computer simulation. When incorporated as a fitness function inside a genetic algorithm this enables the evolution of new microstructures which are stronger and lighter than existing ones. Basanta evolved two-dimensional effector automata to optimise the closeness of particle distribution to some target pattern.

In 2004 Hallinan used a genetic algorithm to create update rules for Reil's artificial genome model, which is a genetic representation that is more detailed than, but functionally the same, as random boolean networks [172, 354]. The update rules are optimised by applying a fitness evaluation function that maximises both the number and length of different limit cycles in order to generate repeating patterns.

In 2004 Suzudo used a genetic algorithm to evolve rules for a two-dimensional asynchronous cellular automaton that, when initialised in a random initial state, will converge into recognisable patterns of columns, a chequered board, and zebra-like stripes (figure 6.17) [423]. The statistical distribution of the cell update rules that lead to successful formation of similar patterns was later examined , and it was found that whilst certain transitions are dominant and appear in many rule sets, the interaction between different transition rules is unpredictable and does not result simply from grouping individual rules together, i.e. the global pattern forming behaviour is an emergent property of the individual rules [423].

6.13 Evolving morphology

The evolution of static morphologies (without control systems) has been an active area of research. Early systems were inspired by the developmental pattern generation properties of biological embryogeny. Later systems have utilised developmental encodings to create functional and aesthetic 3D structures in simulation, and successfully transfer



Figure 6.17: *Evolved asynchronous cellular automata that, from a random initial state, generate three different globally recognisable patterns: checkered, zebra-like stripes, and columns.*

Credit for image: Tomoaki Suzudo [423, 424]

these designs to reality. The focus of morphological evolution has often been to create novel designs that humans would be unlikely to consider. This can be seen in its use for architectural modelling in which interestingly shaped buildings are created, and in the development of everyday objects such as tables and chairs.

The closely related field of “evolutionary art” involves the creation of images, animations, music, and physical objects with genetic algorithms [220]. Manipulation of the genomes of real animals has also opened up the field of “living” genetic art [448]. These fields emphasise visual and auditory stimulation, and human aesthetic appraisal, rather than practicality. There is some intersection with the field of evolving morphology when geometries and objects are created. Due to the focus on creation of aesthetically pleasing designs, and the difficulty of creating fitness functions for this task, evolutionary art often uses interactive evolution, in which a human participant manually selects parents for every generation.

6.13.1 Timeline

Lindenmayer systems were first proposed by the Hungarian biologist Aristid Lindenmayer in 1968 as a model for the development of plant cells [260]. Lindenmayer was attempting to create a mathematical abstraction of the developmental process of cell mitosis and differentiation he observed in algae and flowering plants. With the advancement of technology and the development of computer systems he was eventually able to simulate and visualise advanced plant developmental processes in 3D twenty years later (figure 6.18) [345]. Lindenmayer systems are often used in evolutionary



Figure 6.18: *Three-dimensional plant structures created using manually designed Lindenmayer systems.*

Credit for image: Przemyslaw Prusinkiewicz [345]

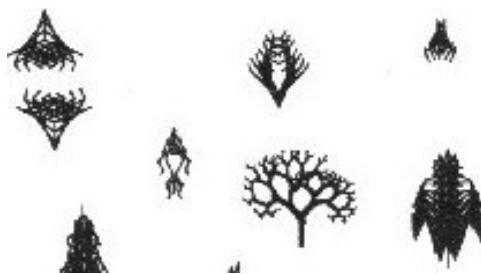


Figure 6.19: *Biomorphs, the first creature morphologies to be evolved.*

Credit for image: Richard Dawkins [85]

morphology, in which the production rules that form the expansion grammar are encoded into a genotype. For an overview of the field of evolutionary Lindenmayer based morphologies see [285].

A Lindenmayer system consists of functions that repetitively rewrite a string based on various rules of grammar. For example, a rewriting rule may be $(A \rightarrow AA)$, in which the single character A is replaced with AA. When applied to a seed string “A” it will result in the string “AA”. When applied again, the string becomes “AAAA”, and so forth. There are variations on this system that change minor features, such as using parametric rewriting rules, but all retain the basic idea of an iterative process of symbolic expansion leading to a final string consisting of some sequence of final symbols. The set of rewriting rules creates a formal grammar of the sort familiar to computer scientists.

In 1985 Dawkins released the first software program to carry out interactive evolution [85]. He termed the evolved creatures “biomorphs” (figure 6.19). Parents were chosen by interactive selection, and reproduction was by process of mutation only.

In 1995 Bentley presented the evolution of complete tables from scratch v [22]. He

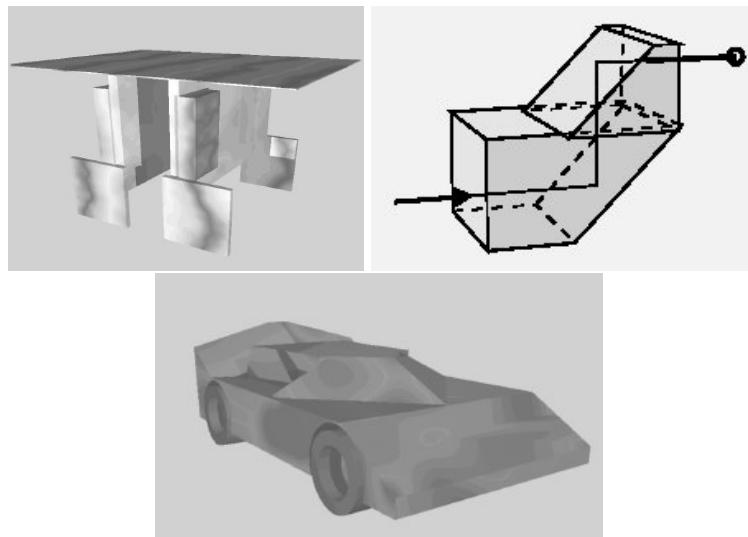


Figure 6.20: *Evolved geometries: table, optical prism, and sports car. All are assembled from simple geometric primitives directly encoded in a genome.*

Credit for image: Peter J. Bentley [22]

used a genotype that encoded variable sized spatial partitions and their locations in 3D space. Since this genotype allowed overlapping geometric primitives to occur, which was forbidden in the phenotype, the morphogenesis function had to prevent or remove overlapping segments. The fitness function aimed to maximise tables that were human sized, of low mass, possessing high stability, and with a flat surface. Many designs were successfully evolved (figure 6.20), although the semi-optimal human design with four legs and a flat cuboid surface did not appear, suggesting that either the search space was too large, the fitness function was deficient, or the genotype representation was inadequate.

In 1996 Bentley presented further work evolving heat sinks, optical prisms, streamlined boat hulls, and cars [20]. All of these designs aimed to optimise some functionality, such as minimising air resistance, or multiple factors like maximising surface area whilst minimising volume. All of the composite geometries were assembled from simple geometric primitives directly encoded in the genome.

In 1997 Eggensberger produced the first morphologies to be developed using a biologically realistic cell based embryogeny (figure 6.21). He simulated genes, cell mitosis and death, and gradiated environmental marker molecules. The fitness function selected for morphologies with bilateral symmetry and distance from some arbitrarily chosen desired number of post-development cells.

In 1997 Funes developed the first physics based morphological evolution system,

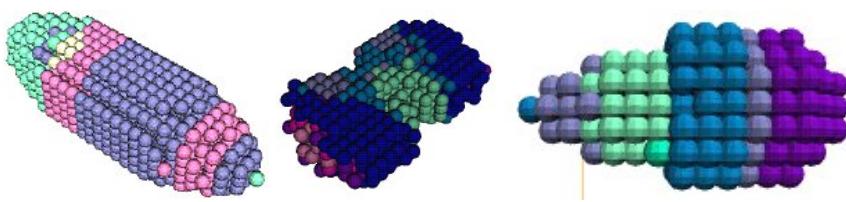


Figure 6.21: *Morphologies evolved from a single cell using a complex biologically realistic developmental embryogeny.*

Credit for image: Peter Eggenberger [120]

and showed that the evolved morphologies could be successfully transferred to reality by construction from Lego blocks [143]. The genotype encoded a tree structure, where each node represents a single Lego block, with four possible child nodes representing the four corners (viewed sideways) of the block. The physics simulator was two-dimensional, so the encoded genotypes were 2D, and the Lego blocks used were of unit width but arbitrary length and height. Each parent-child joint encoded the amount of overlap, which affects both the strength of the joint and the distance the child extends beyond the parent.

Funes successfully evolved two-dimensional weight carrying structures; a bridge with maximised length, a scaffold, and a crane with maximised weight lifting capacity. In 1999 Funes reported successfully evolving a three-dimensional structure (a weight bearing table) which was also transferred to reality by construction in Lego (figure 6.22) [144].

In 1998 Ochoa evolved Lindenmayer systems with a genetic algorithm to create two-dimensional tree like structures (figure 6.23) [322]. Parent selection is either interactive, allowing for the selection of individuals and features that are aesthetically interesting, or can be based on a fitness function that attempts to evaluate an individual's ability to collect light based on important factors, such as the ability to grow vertically and maintain stability whilst having a high degree of branching in order to maximise exposed surface area.

In 1999 Rosenman presented an approach to evolving modular designs for two-dimensional morphologies [367]. He was concerned with generating architectural floor plans. The fitness function optimised for properties such as minimising overall wall length, having some desired number of rooms, and maximising room size. The genotype was a control program for a “turtle” operating on a 2D substrate, with individual genes specifying operations such as turning and moving.

Also in 1999 Taura released details of a system for creating 3D layouts for satellite

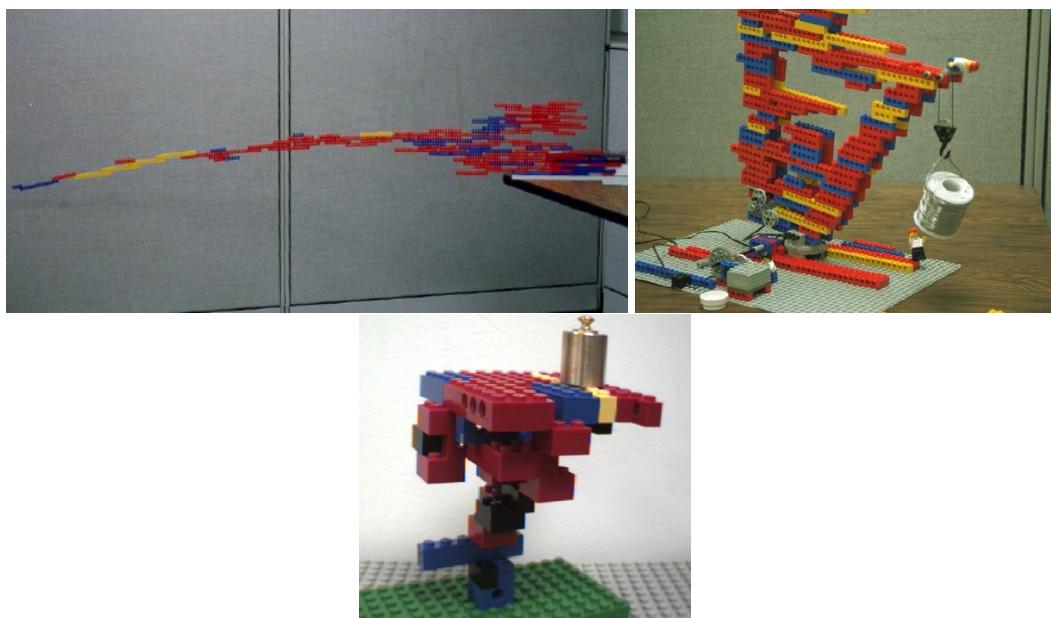


Figure 6.22: This bridge and crane were built with Lego bricks according to a two-dimensional structure evolved in a genome. The three-dimensional table was later evolved. The fitness functions evaluated length, in the case of the bridge, and weight bearing capacity, in the cases of the crane and table.

Credit for image: Pablo Funes [143, 144]

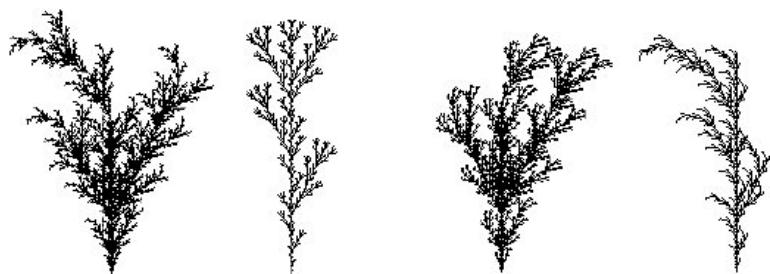


Figure 6.23: Two-dimensional plant structures evolved using Lindenmayer systems.

Credit for image: Gabriela Ochoa [322]

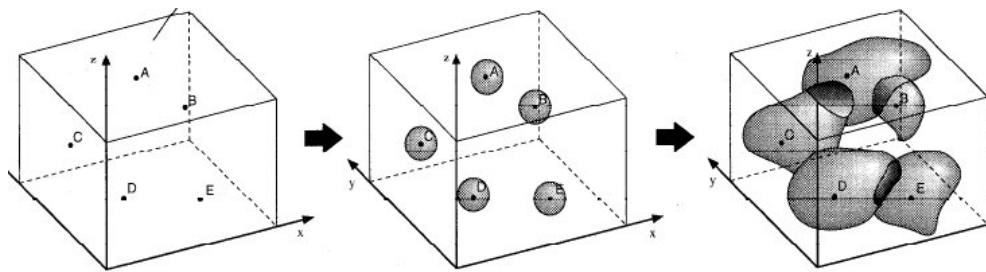


Figure 6.24: *Cells divide and form shapes to pack the internals of a satellite.*

Credit for image: Toshiharu Taura [433]

design [433]. Satellites are very space constrained, so designers must invest a lot of time in packing modules into the available space. This is complicated by the fact that many modules do not have preformed shapes; although their rough dimensions or volume may be inferred from their function, the actual shape can be designed around the shape of other components within the satellite. Taura designed a 3D biologically based developmental encoding that simulated cell division, migration and expansion into arbitrary morphologies (figure 6.24). The fitness function measured how well all the components fit the known constraints, and minimised overlaps.

The evolutionary process attempted to create high fitness modular blocks of genes and then utilise them as genetic building blocks. This was done by looking for sequences of genes that occur in individuals with an above average fitness, and then adding or removing these sequences in samples, and re-testing to confirm the hypothesis that the sequence increases fitness. If true, the sequence is frozen and made accessible to the evolutionary operators as a single block (just like an individual gene). Rosenman observed that genome length increased linearly with phenotype complexity when evolving hierarchy, versus exponentially with no hierarchy.

In 1999 Bentley evolved tessellating two-dimensional grid based morphologies [23]. He compared the performance of three different functions for morphogenesis, which he termed external, explicit and implicit. The external genotype encoded a sequence of 2D primitives which had been manually designed. The explicit encoding used a genetic program tree which, starting with a seed cell, could be used to develop the phenotype. The implicit encoding used a cellular automaton with evolved rules; Bentley actually claims that this is not a cellular automaton since there are no updates for white cells, but clearly the updates are deterministic and based on neighbouring cell states, so this is equivalent to a cellular automaton with a fixed transition rule for white cells. Bentley showed that only the implicit embryogeny could evolve perfectly tessellating

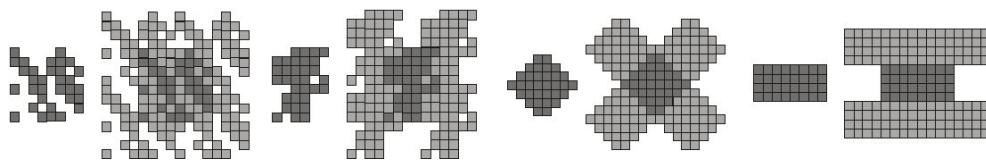


Figure 6.25: *Two-dimensional patterns are used to build larger tessellating morphologies to compare systems of developmental embryogeny. Each tile is shown along with its larger pattern. Embryogenies from left: none, external, explicit, and implicit.*

Credit for image: Peter Bentley [23]

tiles for all of the grid sizes under test. Figure 6.25 shows the evolved tiles and their use in a larger tessellating pattern for each embryogeny type.

In 2000 Bentley demonstrated the first evolution of gliding morphologies [21]. The evolved shapes might be more accurately described as “falling” morphologies, as, like a sycamore seed, the fitness function is proportional to the length of time between an object’s release from a fixed height, and it coming into contact with the floor. The genotype encoded the shape of an arbitrarily sized polygon, which was printed, cut out, and tested in the real world.

In 2001 Hemberg produced the first evolutionary system for 3D map Lindenmayer systems [186]. Map Lindenmayer systems specify a graph based grammar that operates on a two-dimensional surface. This was extended to 3D to allow the production of 3D surfaces, and the production rules encoded in a genotype. The fitness function measured distance of the individual from a variety of user specified ideal values, for a variety of properties, such as size, smoothness, respect of boundaries, amount of surface division, and symmetry. Various surfaces were evolved. In 2004 Hemberg reported that several designs had been physically manufactured, including a pneumatic strawberry bar [188]. In 2006 the results of several architectural projects using Hemberg’s system were presented (figure 6.26) [187].

In 2001 Hornby also produced a Lindenmayer based evolutionary system for 3D designs [198, 205]. He used it to evolve a variety of tables, and directly compared a Lindenmayer encoding with a direct encoding, showing that the generative Lindenmayer system produced populations with significantly higher fitness. Visually, there was an obvious difference between the two encodings, with the Lindenmayer based designs being highly regular and symmetrical (figure 6.27). The designs were transferred to reality using a 3D thermoplastic printer (as described on page 203). In 2005 Hornby compared different genotype representations for the table task and showed that

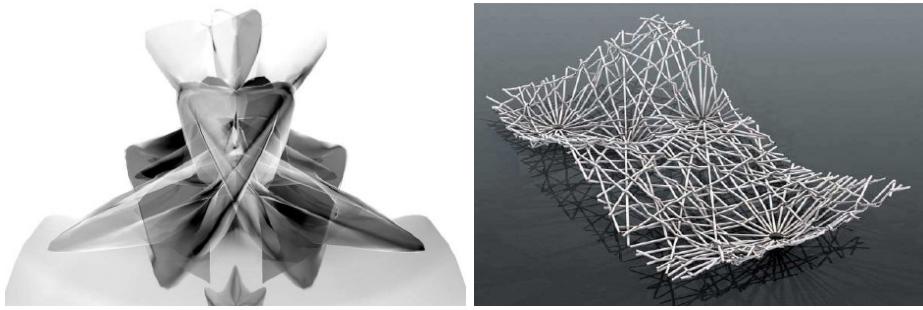


Figure 6.26: *An evolved architectural three-dimensional surface design (left), and another with 90 nodes and 1000 joints that has been fabricated in reality (right).*

Credit for image: Martin Hemberg [187]

the best encoding utilised and exploited modularity, regularity, and hierarchy [203].

In 2002 Ebner evolved 3D Lindenmayer system based plants [119]. Plants were grown from an initial seed following the evolved production rules. The fitness function judged plants according to their light capturing qualities in order to encourage development of a natural morphological form.

Also in 2002 Thomas used an interactive developmental evolutionary system to evolve arbitrary 3D meshes which are texture mapped and rendered with genetically determined features [438]. Each morphology is developed from an initial seed to which various transformations are applied (figure 6.28). Interactive parent selection was used since the evaluation of visually appealing characteristics is a subjective task.

In 2003 Ebner evolved the morphology of wind turbine blades [118]. A full 3D simulation of a three blade wind turbine was implemented using the “Open Dynamics Engine” physics simulator. The effect of wind was simulated using many particles to which a force due to wind was applied at every time step, and collisions of these particles with the surface of a turbine blade generated a small contact force which helped drove the turbine around. The fitness function was proportional to the rotational velocity of the turbine blades. A variety of blade designs were successfully evolved (figure 6.29), the appearance of which is reminiscent of human designs.

In 2004 Miller evolved Cartesian genetic programs that develop in a two-dimensional grid to form a French flag pattern [299]. The program is a digital circuit that maps input chemical and cell signals to output chemical and growth signals. A “(1+4)-ES” evolutionary strategy was used, in which one parent is chosen from a population of five, four children are generated by mutation, and both parent and children are placed into the next generation. Fitness is assessed by how closely the developed phenotype

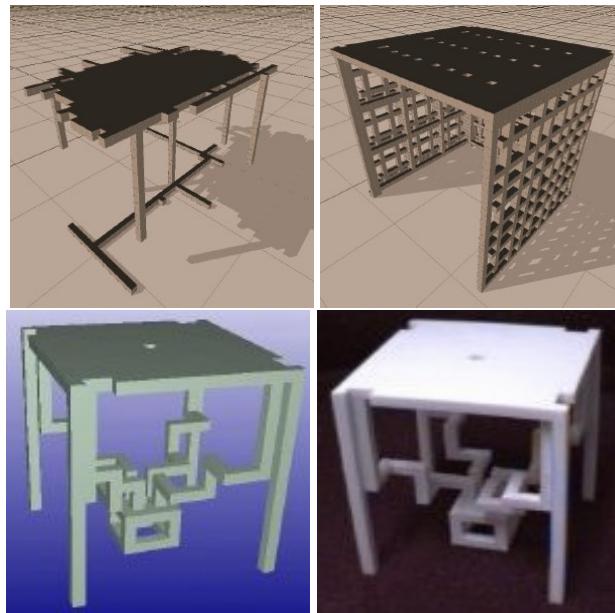


Figure 6.27: *Evolved table designs. Typical example of a direct encoding (top left), Lindenmayer encoding (top right), and a table with its thermoplastic counterpart (bottom).*
Credit for image: Greg Hornby [198, 205]

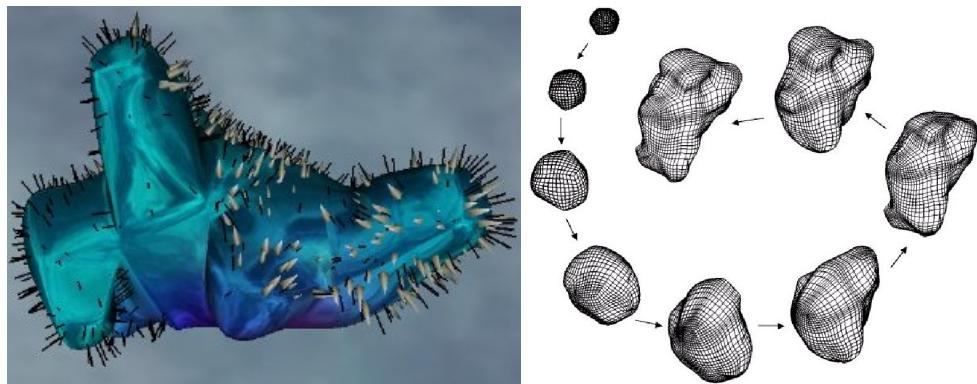


Figure 6.28: *A variety of transformations and visual effects are interactively evolved and iteratively applied, beginning with a unit cube seed, to create smooth, textured, colourful 3D objects.*
Credit for image: Dale Thomas [438]

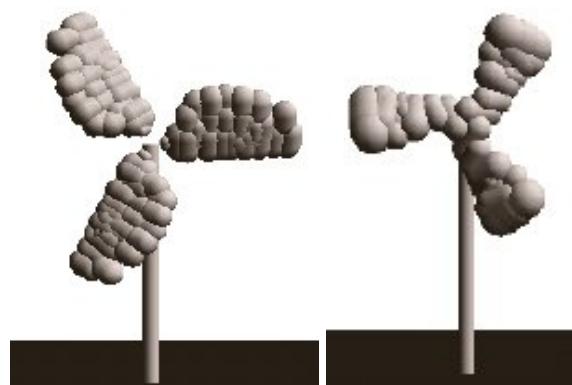


Figure 6.29: *Turbine designs with blade morphologies evolved inside a physically realistic particle simulator.*

Credit for image: Marc Ebner [118]

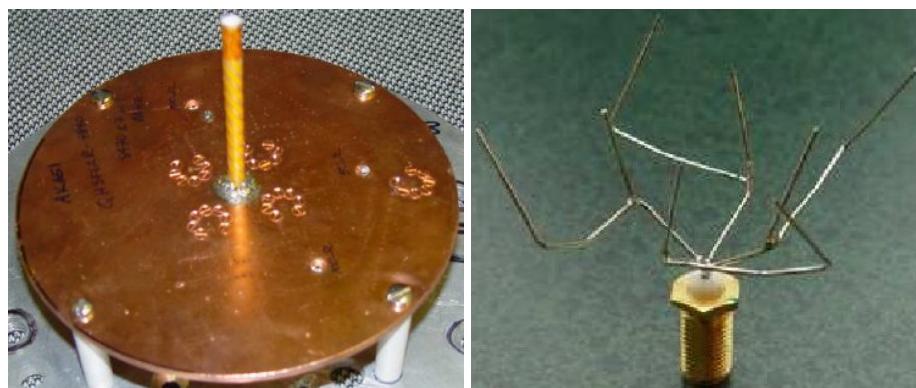


Figure 6.30: *A human designed QHF antenna (left), and an evolved antenna (right). The evolved design is not only radically different, but also performs better under a range of criteria.*

Credit for image: Jason D. Lohn [264, 265]

represents the French flag; the size is specific, so the cells must discover some method of modulating growth once the correct size is reached.

In 2004 Lohn used a genetic algorithm to evolve the morphology of a radio antenna, which was manufactured and used in a real space mission [264, 265]. The evolved antenna has an unusual shape, which is radically different to a human designed antenna (see figure 6.30 for a comparison). It was reported that the evolved antenna was superior to the human design in several ways; it had a higher gain, did not require matching electrical circuitry, produced a more uniform signal across a wider range of elevation angles (the angle of the satellite relative to the surface of the Earth), and required less effort to design.

In 2005 Rieffel evolved “buildable” designs [360, 361]. He was concerned that the manufacturability of designs in the real world was not usually considered when designing genomes and developmental systems. Previous works had evolved recursive, modular encodings capable of representing designs consisting of many different components. However, the output of the developmental process was a complex design with many interacting parts, and no assembly instructions. Considerable human time was expended during the construction phase figuring out how to interlock the various complicated 3D geometries of different parts in order to build the final design.

Rieffel emphasised two things that were necessary for ease of construction in the real world. The first was to use a genotype encoding an “assembly plan”, which specifies exactly how the object is to be built and assembled from its sub-components, rather than just encoding the object itself. The second was to use “situated development”, in which developmental morphogenesis occurs in the same area in which fitness is evaluated. For example, in the previous work on evolving Lego based morphologies, rather than have a Lindenmayer system manipulating the position and orientation of Lego blocks in an abstract space, it would instead manipulate sequences of building instructions, and the building phase should be carried out inside the same 3D physics simulator used for fitness evaluation, with a small element of noise. These constraints would ensure that there was a clear assembly path which could be followed in a 3D world, constrained with real physics, and which would be robust to minor deviations.

Rieffel’s target application was automated 3D manufacturing machines. The design was carried out in a simulated 3D world (using *Open Dynamics Engine*), with a “turtle” type developmental program that could move in the X and Z planes and deposit scaffold or permanent bricks (scaffold bricks are automatically removed after the building phase, before any fitness evaluation). A multiple-objection optimisation function was used, which analysed the cross-section of the three-dimensional building along the XZ plane inhabited by the turtle, and rewarded structures for providing a large overall area, a large open area beneath or within the structure, a small assembly plan, and for using fewer bricks. The evolutionary algorithm ended up creating archetype structures joined by a connecting roof, with a similar appearance to the temples of classical Greek architecture.

In 2005 Preble evolved morphologies for two-dimensional photonic crystals [343]. Photonic crystals use nano-tunnels that act as filters to remove or amplify light at specific frequencies. This mechanism is used in the wings of butterflies to create a stunning visual effect. Synthetic crystals can be grown by humans into any desired shape.

However, calculating the necessary factors to end up with a final arrangement is a difficult task, as the equations are complex with many interacting variables.

Preble used a genetic algorithm to evolve both bitmaps and tree structured genotypes that can be used to create a phenotype lattice. The tree based genotype produced higher fitness individuals, and in both cases performance was significantly better than individuals created by random search. The best evolved individuals exceeded the best human design by 12.5%. Analysis showed that the evolved individuals used a honeycomb shape similar to that found in the photonic crystals of living creatures. Nature has already optimised this arrangement through biological evolution, so it is unsurprising that an attempt to replicate this process would end with a similar design.

In 2006 Stanley observed that, to date, the encoding of morphological genotypes had failed to capture the essence of living lifeforms [410]. This was attributed to several failings; the lack of limits to recursion producing overly fractal-like designs, the existence of perfect regularity and symmetry which never occurs in nature, and the inappropriate coding of modularity which allows bizarre mutations that would never occur naturally. Stanley argued that systems of encoding should aim to capture, and be indistinguishable from, natural biological characteristics.

In 2007 Stanley evolved “compositional pattern producing networks” (CPPN), which are neural networks that can be used to construct two-dimensional grid based morphologies [411]. An interactive genetic algorithm was used to evolve feed-forward neural networks with the “neuro-evolution of augmenting topologies” (NEAT) algorithm [409, 413]. Two-dimensional grayscale images can then be assembled a pixel at a time by using the x, y coordinates of the pixel as inputs to the neural network, and interpreting the single output as an intensity level. The function evaluated by the neural network hence implicitly encodes the whole output image. A similar scheme was used by Hastings to evolve neural networks which in turn produce 3D particle effects for video games [182]. Figure 6.31 shows some of the evolved images.

CPPNs are markedly different from other methods of morphological construction. No other system uses evolved neural networks to generate morphologies. Other systems tend to use developmental encodings that grow solutions starting from a single seed, so no individual part of the final morphology can be constructed without constructing the whole individual. CPPNs instead allow the state of final cells to be evaluated individually.

The use of a grid based system is also unique; other systems utilise geometries with cells, or other primitives, occupying points in 2D or 3D space. Instead, the CPPN

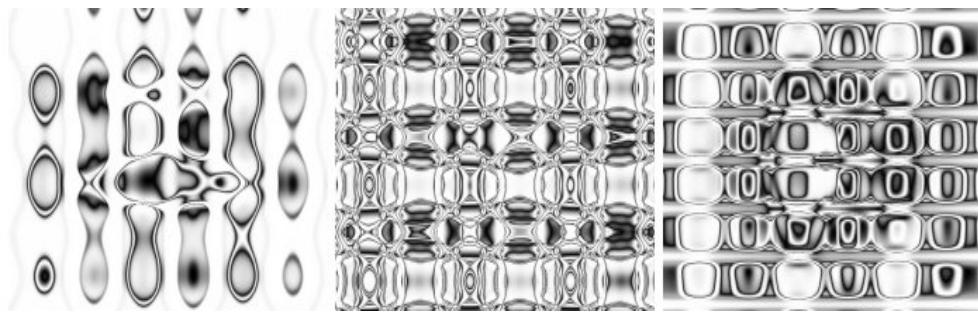


Figure 6.31: *Despite being generated from a neural network in a pixel by pixel scan these evolved images show features of regularity and modularity typical of developmental encodings.*

Credit for image: Kenneth O. Stanley [411]

system uses a fixed width grid, which has the disadvantages of being quite rigid, with a fixed resolution, and quite difficult to simulate, since complex structures consisting of many cubic primitives can form. The use of pixels or voxels means the computational and storage requirements for the phenotype may be significantly increased over other representations, even though the genotype is relatively compact.

An advantage of the CPPN system is that its use of neural networks expands the range of evolutionary techniques available, allowing the large body of existing research on neural network evolution to be utilised. The NEAT algorithm is one of the best performing evolutionary algorithms for neural networks. It works by evolving both weights and topology, and reproduction operators are designed to preserve the function of the parent network whilst allowing minor variations, i.e. to allow mutations to augment the existing functionality, rather than disrupt it, providing a clear evolutionary path from small, unspecific networks, towards larger, more complex ones. It has been proposed that it may be possible to evolve any kind of phenotype using NEAT, although for many problems it is difficult to see how the continuous function calculated by the neural network could be transformed into a valid phenotype.

6.14 Evolving robot morphology and control

The argument has been made by several authors that the combined evolution of body and brain allows the genetic algorithm more flexibility in the selection of working solutions, and allows good solutions to exploit the synergy between control and morphology [50, 67, 257, 321]. This synergy is more likely to arise if both the brain and

morphology can adapt simultaneously, utilising an evolutionary path that allows many small adaptations to be taken in concert. An analysis of exactly how this synergy occurs concludes that co-evolution constructs extra-dimensional bypasses in the solution fitness space that allows the search to move through regions between fitness peaks that would not have previously been accessible [39].

The DNA of living creatures contains genes that code for the construction of both the body and brain. Thus from nature we have a proof by existence that it is possible for body and brain to be evolved together to reach human levels of intelligence. It is unlikely that the evolution of a brain for a pre-existing fully formed human body would succeed; the space of input sensory data and output actuators is enormous, and it would be a computationally intractable task to randomly search the space of controller mappings between input and output to find a successful match. In contrast, evolution starting from simple models allows both control and morphology to adapt to each other, and enables the hierarchical and modular composition of building blocks which narrow the search space.

Living creatures display a tight coupling between morphology and control. Many specific behavioural traits controlled by the brain are only made possible when the morphology has specific corresponding features. The morphology of living creatures can perform a variety of computational functions [332], which suggests that it would be inappropriate to evolve morphology in isolation from the control system, or vice versa. The adaptation of morphology through evolution can also work to reshape the search space and fitness landscape, making the evolutionary process more efficient and likely to succeed [363].

There are two approaches to the evolution of virtual creatures [261]. One is to provide some kind of existing solution, or partial solution, and allow the genetic algorithm to adjust morphological parameters and evolve a control system. This tends to be viewed as optimisation. The alternative is “open-ended synthesis”, where solutions are evolved from scratch, and the user has no particular solution in mind when the genetic algorithm is deployed.

6.14.1 Timeline

Several researchers have co-evolved robot control systems and morphologies to create simulated, and in some cases real, robotic systems. Note that this timeline is not in a precise order; some attempt has been made to keep threads of research by the same

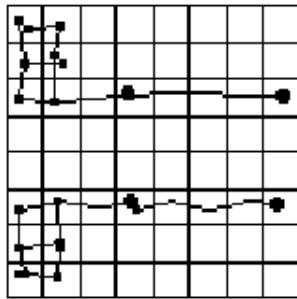


Figure 6.32: *An agent in a grid world created by development of an initial seed with a boolean network genome, with different nodes coding for cellular and neural development. This agent was manually designed and then optimised with a genetic algorithm.*

Credit for image: Frank Dellaert and Randall Beer [98]

authors sequential in order to increase readability.

In 1994 Dellaert and Beer released a technical report detailing their work on the co-evolution of body and brain for autonomous agents in a 2D world [98]. They implemented a developmental environment based on the simulation of gene regulatory networks by evolved boolean networks. Starting from an initial seed, the development of a cell was determined by the functioning of the boolean network. Nodes in the boolean network represent the synthesis of different molecules. Threshold levels of molecules determine when the cell undergoes mitosis, differentiation, inter-cell signalling, and the growth of neural connections.

Dellaert and Beer showed that their system could express complex morphologies by hand crafting a robot and neural network design (figure 6.32), and then showing that a genetic algorithm could successfully optimise it. At this time their attempts to evolve complete agents from scratch were unsuccessful, although they succeeded two years later [99].

Later in 1994 Sims presented his work on the evolution of morphology and control for creatures inside a simulated 3D world (figure 6.33) [397, 398]. Sims used a single genome for both the brain and body, and managed to successfully evolve creatures from scratch that performed a variety of tasks. This was the first work to combine evolution of morphology and control in three dimensions.

Sims used a developmental coding for the genotype. The body was represented as a cyclic graph, with each node specifying the construction of a body part. One body part was randomly chosen as the root node, which was used as the initial seed for growing the rest of the body. During morphogenesis the graph was unrolled, starting from the

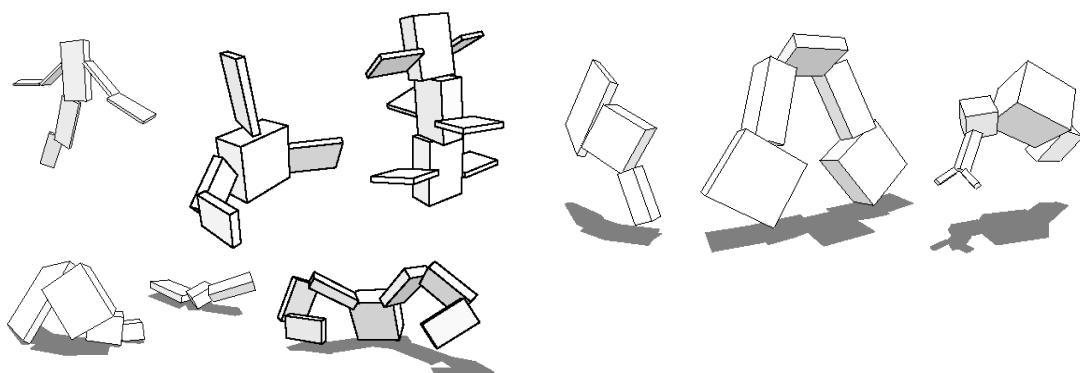


Figure 6.33: Sims' original work produced creatures with evolved control networks and bodies that could swim (top left), jump (right), and walk (bottom).

Credit for image: Karl Sims [397]

root node, and following all paths, in order to remove cycles and convert the graph to a tree. This tree was then be converted into a sequence of geometrical primitives (in this case, cuboids) linked with joints, which imposed movement constraints on the pairs of bodies they connected (figure 6.34).

The evolved control system consisted of a neural network for each body part, with a limited amount of connectivity between the networks of neighbouring body parts. The formation of a central “brain” was encouraged by creating a single network which could link to any of the localised body part networks. The neural network model was not the classic sum-and-fire used by other works, instead each node could perform one of several mathematical functions, or could generate periodic waveforms of various shapes at different frequencies. This gave the evolutionary process the ability to easily generate networks with oscillating outputs, which underlie cyclic joint movements, and therefore could be expected to hasten the development of successful agents.

Sims used his own physics simulator based on a Runge-Kutta integrator. The fitness of creatures was evaluated based on their performance on several tasks, including movement across land (maximising distance travelled), swimming (where the bodies are subject to reduced gravity and viscous drag), jumping (maximising instantaneous height), light following (with appropriate sensor), and a game of “block grabbing”, where two creatures were placed into an arena, opposing each other with a block between them. The creatures gained points for touching the block, and were penalised when their opponent touched the block.

Creatures were successfully evolved for all of the tasks. The creatures, affectionately termed *Blockies*, could move, swim, jump, sense light, and fight. The co-

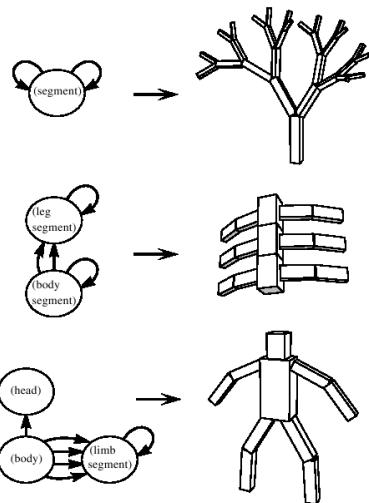


Figure 6.34: *A morphogenesis process for body part graphs. Each edge is followed, starting from the root, in order to build the phenotype. Nodes map to individual body parts. The genotype combined features of modularity, reuse and symmetry that were later shown to be advantageous to evolutionary success.*

Credit for image: Karl Sims [397]

evolutionary block grabbers evolved strategies to simultaneously gain possession of the block and shield it from the other agent, much like a rugby player.

Sims's novel work had an inspirational effect on the field of co-evolutionary robotics, which has been recognised in the subsequent work of others [60, 276, 295, 348, 393, 436]. In particular, his use of 3D, and the publishing of short rendered movies of the evolved creatures was unique at the time, and gave a powerful visual emphasis to his achievements.

In 1995 Ventrella presented his system for evolving “funny animated figures” (figure 6.35) [465]. These figures, which were initially evolved in 2D and then 3D, have stick like bodies connected by multiple degree of freedom joints. There are two control systems, one for bodies with fixed numbers of joints, and one for bodies with variable numbers. The former use sine wave generators connected to each joint, with the frequency, amplitude and phase offset being evolved. Since Ventrella wanted to use a fixed size genome the latter control system, having varying number of joints, uses a parametric sine wave series generator, with the parameters for this generator being evolved.

In 1996 Lee co-evolved parameters of the morphology of a wheeled robot (body size, wheel radius, and wheel base), along with a genetic programming based control

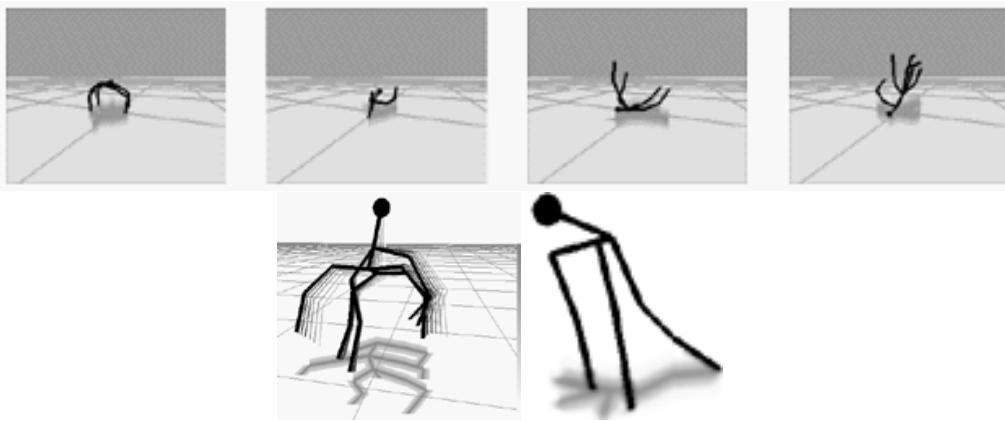


Figure 6.35: Examples of several virtual creatures evolved for locomotion. Each body is evolved from a tree topology genotype, and joints are directly controlled by sine wave generators.

Credit for image: Jeffrey Ventrella [465]

system [257]. The robot was simulated in a 2D world. The robot was equipped with infra-red sensors, and was evaluated on an object avoidance task. He demonstrated successful co-evolution of the genetic program and morphological parameters. The morphological parameters were then varied, showing that the genetic algorithm had found an optimal body for the control system, and that they had been intrinsically linked by the evolutionary process.

Also in 1996 Cliff evolved sensory-motor morphologies and neural control for 2D circular robots in simulation [72]. Two populations were co-evolved in a pursuit and evasion game. It was shown that the evolution of each species acted as a powerful force on the other, and that the evolution of innovative strategies within one species would be quickly counteracted by the other.

In 1999 Komosinski first presented the beginnings of his “Framsticks” evolutionary system [239,243]. Like Sims’s work, it allowed the user to evolve both the morphology and neural network based control system of virtual creatures (figure 6.36). The system also provides a morphology editor, which a human designer can use to handcraft a robot morphology, with sensors and muscle actuators. A neural network control system can be evolved following the development of the morphology.

“Framsticks” morphology is based on straight sticks connected with elastic joints. The simulated physics bodies and the joints occupy the same points in space, at the spherical ends of each stick. The elasticity of the joints provides some element of morphological computation by stretching and dampening movements due to instantaneous

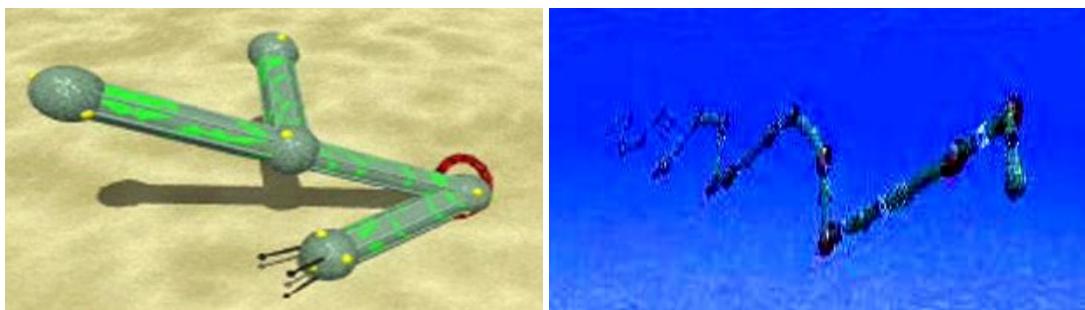


Figure 6.36: *Virtual creatures with evolved morphology and neural control in the Framsticks world.*

Credit for image: Maciej Komosinski [239, 240, 243]

forces. There are various models of sensor and motor neuron which interact with the morphology. Visualisation of the various sensors is possible as they are rendered as a differently patterned or shaped spherical stick-end.

The physics simulator, “Mechastick”, was written specifically for Framsticks, and had the usual movement, gravity, and friction models for land, and fluid drag and resistance models for water. Unusually for 3D physics based simulations, the simulator allowed closed loops to be made in the morphology, as opposed to the usual tree structures.

The neural networks were not traditional models; the nodes could perform the standard sigmoid of the weighted sum of inputs function, but could also generate sine waves, random noise, compute series difference, and pass through signals with arbitrary delays.

Sensory perception was via gyroscopic receptors, distance sensors focused on specific objects, smell, and energy source (or food) sensors. The energy source sensors are necessary to enable virtual world simulations, where many different agents compete against each other to obtain energy, with physical activities having a cost in terms of energy expenditure [240]. Smell sensors allow agents to discriminate between other agents, and between different classes (families, or ecological niches) of agent. It is possible to set up an environment in which robots can fight and kill each other, with the bodies of dead robots releasing energy, thus providing an evolutionary incentive towards survival of the fittest.

Three basic genotype encodings were implemented [240]. The basic direct encoding, known as “simul”, is simply a list of body parts and their attributes. There is a one-to-one mapping between genotype body parts and phenotype body parts, so it can

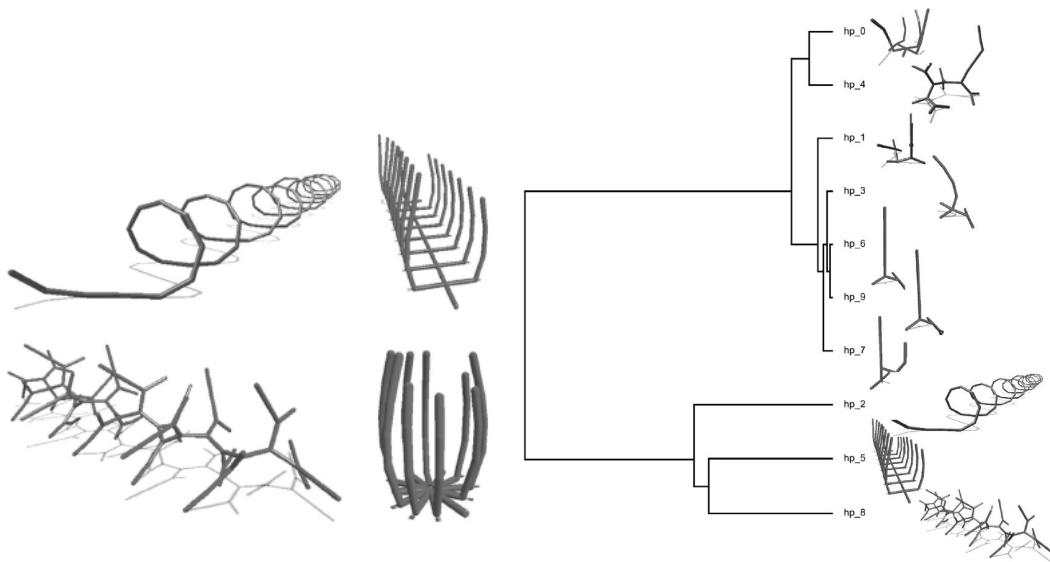


Figure 6.37: *Passive Framsticks agents evolved to maximise static height, and an automatically generated phylogenetic tree, showing the common ancestry of different agents.*

Credit for image: Maciej Komosinski [241, 242]

represent any possible phenotype in the world. The recurrent direct encoding “recur” is the same, except that body part attributes can be reused by different body parts. Finally, the indirect developmental encoding “devel” supports the reuse of body part modules, which necessitates a recursive developmental process.

These encodings were directly compared on three fitness tasks (passive height, active height, and velocity) in 2001 [242]. It was shown that the best individuals for the passive and active height tasks were those with the developmental encoding, although the direct recurrent encoding beat it on the velocity task.

In 2003 Komosinski presented a method for the development of fuzzy controllers for the simulated agents, and a method for automatically clustering similar genotypes into phylogenetic trees (figure 6.37) [241].

In 2000 Bongard developed a system of 3D morphological and neural evolution (figure 6.38) [34]. This system was unique in using large numbers of small spheres, connected with both moving and static joints, in order to construct larger morphological features. Many morphological evolutionary systems are limited to relatively small numbers of physics bodies, as the computational requirements of simulating interconnected bodies increase non-linearly, but apparently this was not a problem for this simulator.

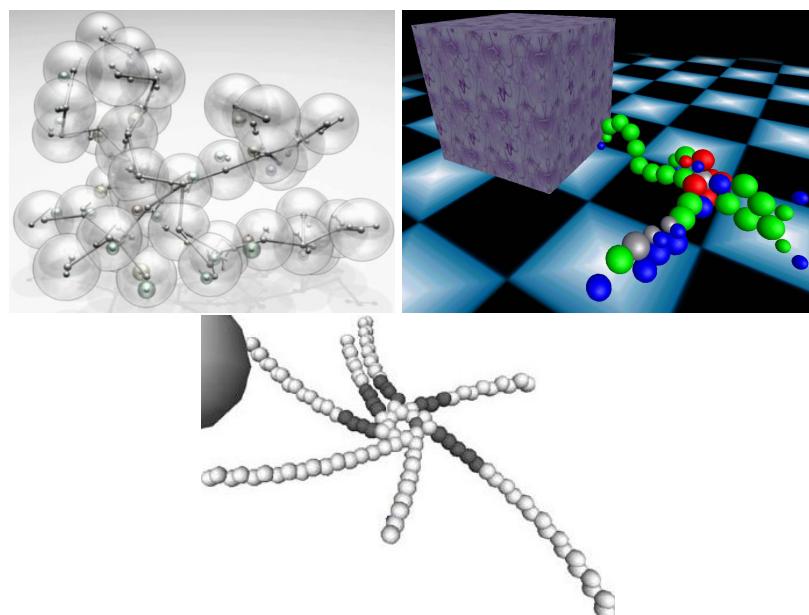


Figure 6.38: *From top left: a transparent agent with embedded neural network, a block pusher, and an agent with biologically plausible morphogenesis.*

Credit for image: Josh Bongard and Daniel Bisig [34, 38]

Bongard's system was used to compare the importance of enforced symmetry in the development of morphologies and neural connections by evaluating path following and metabolic activity for a locomotive task. He showed that symmetry was an important factor in these activities; asymmetric agents were slower, less efficient, and less capable of following a straight path. In 2002 he used the same system to show that agents from successful evolutionary runs were more likely to have modular genetic regulatory networks than agents from unsuccessful runs, suggesting that genetic modularity is a beneficial trait for evolutionary development [37].

In 2003 Bongard used a more biologically realistic morphogenesis process to evolve agents for a block pushing task (figure 6.38) [40]. He also showed (perhaps unsurprisingly) that similar, but not identical, genotypes that diverge early in the developmental process have larger behavioural and phenotypic differences than those that diverge later, suggesting that earlier mutations are more significant. The same year, Bongard showed that his biologically based system could evolve genotypes that rely on environmental factors in order to differentiate and self-organise cells during development [38]. Analysis of gene distribution in the evolved creatures showed separation between genes responsible for morphology and neural network control, suggesting development of body brain separation and a degree of canalisation.

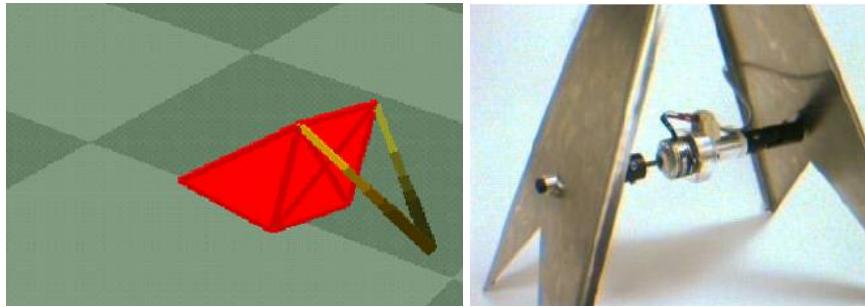


Figure 6.39: *Crossing the reality gap with evolved control and morphology (left) was heavily constrained by manufacturability (right) in 1999.*

Credit for image: Jordan Pollack [338]

In 1999 Pollack *et al.* briefly mentioned, in a paper on general 3D simulation and evolutionary techniques, that they were working on simulation of buildable robots and coevolution of dynamic controllers, with transfer to reality being one of the stated goals [338]. This would be a significant advance - although people had constructed evolved static morphologies in the real world, no-one had successfully transferred an agent with both evolved morphology and control. The main reason for this was the difficulty in building arbitrary robot designs in the real world, although Pollack noted that even Sims's 1994 work on coevolution had yet to be reproduced due to the lack of a versatile and widely available 3D physics simulator. Pollack's proposed designs of both the simulated and real robots were incredibly simple, being highly constrained by manufacturability (figure 6.39).

In 2000 Ray and Taylor both reported that they had independently reimplemented Sims's work on evolving virtual creatures [348, 435]. Ray had approached the problem from the position of creating artificial art, and had therefore made the selection of parents interactive so that human artists could select for behavioural or morphological features that they found interesting. Directly linking the outputs of neural networks to control morphological features like colour made the simulated creatures more visually stimulating (figure 6.40). Taylor and Massey made only insignificant changes to Sims's design, such as using cylindrical rather than cuboid primitives to construct the morphology. They claimed that the differences between their work and Sims's was "more technical than scientific", but Miconi would later claim that this was an incomplete implementation of Sims's work [295]).

In 2000 Lipson and Pollack presented their "genetically organized lifelike electro-mechanics" (GOLEM) system that enabled evolved designs to be transferred to real-

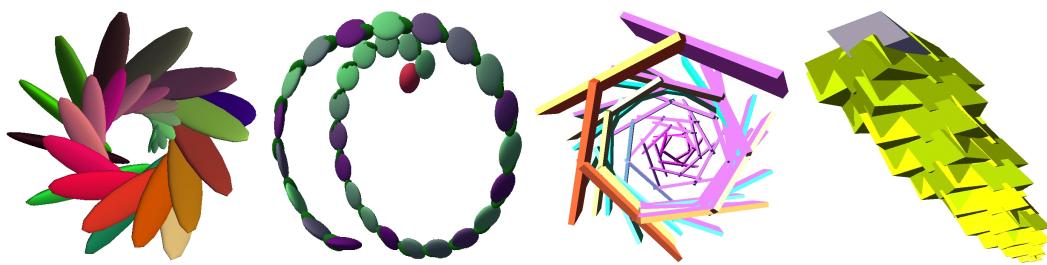


Figure 6.40: *Aesthetically evolved virtual pets. Both morphology and control were subject to human preferential selection via an interactive evolutionary process.*

Credit for image: Thomas S. Ray [348]

ity [262, 340]. Creatures were evolved in simulation with standard genetic algorithms. Neural networks were small, sigmoid, feed-forward designs. Morphologies were constructed from a set of parts including linear actuators and straight cylindrical bars. Joints could be fixed for creating composite structures, or ball-and-socket allowing rotation. The innovation here was in designing a set of phenotype building blocks that could be easily manufactured in the real world using a thermoplastic 3D printer.

Thermoplastic 3D printers are used for rapid prototyping in various fields of design. Thermoplastic is a special material that becomes liquid and deformable when heated, and brittle and hard when cooled. The printers work by melting the thermoplastic, and then depositing it layer by layer through a precision controlled nozzle in a manner similar to an inkjet printer. 3D objects can be directly printed, and even complex structures, such as ball and socket joints, can be easily manufactured.

The printer used in this project was manufactured by Stratasys Inc. and can print any 3D shape within a $8 \times 8 \times 12$ inch volume [339]. Research in the area of 3D printing has great promise; the “Fab@Home” project provides open source designs for hardware, software, and objects, and researchers have reported successful automated manufacture of complex parts, including batteries and actuators [274].

In the GOLEM project, whole robot morphologies were printed as a single plastic object. Linear actuator insertion points were printed as thin struts which could be pushed out by hand and replaced with snap-in motorised actuators. The transfer of these robots to reality was successful. The real robots had the appearance and behaviour of the simulated ones (see figure 6.41 for a comparison). The only concession made was to add noise to the simulator in order to evolve robustness to variation. The neural network controller was simulated on a standard PIC micro-controller attached to the robot.

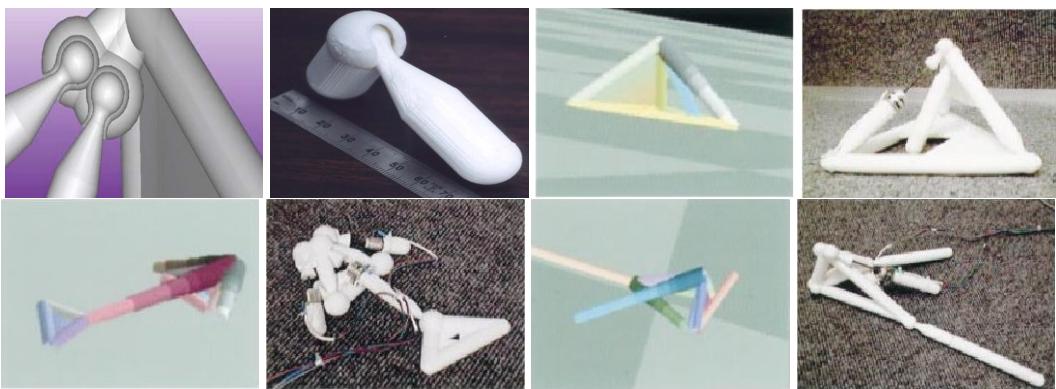


Figure 6.41: Various GOLEM simulated phenotypes and their real world thermoplastic counterparts.

Credit for image: Hod Lipson and Jordon Pollack [262]

Successful transfer to reality in this case may be surprising to some given the limitations of the physics simulation engine, which only simulated quasi-static motion. This means that all objects in the simulator must be statically stable, and must always be in contact with the ground. This model is quite restrictive, although it is used in studies of robotic motion due to the ability to mathematically analyse the equations of motion.

Statically stable movements appear unnatural to a human observer. For example, in biped robots statically stable controllers always keep a foot on the ground, and ensure that the other foot is precisely placed in order to balance the robot at all times. At any point the motion of the robot could be frozen and it would not fall over. The robot is unable to fall through periods of instability as humans do when they walk naturally. The successful transfer to reality in this case shows that statically stable simulation was a sufficient minimalist abstraction as defined by Jakobi [215, 216]. On the other hand, the simulator constraint may well have biased the search towards creatures that are statically stable at all times, which may have had the beneficial effect of eliminating a huge area of the search space where locomotion is more difficult.

In 2001 Hornby and Pollack presented their results from co-evolution experiments on creatures constructed from interconnected straight lines with simple non-networked joint oscillators [204, 205]. Like Pollack's work in 2000, the simulator only simulated statically stable movements, but unlike most physics simulators it was capable of simulating massive numbers of bodies in complex interconnected hierarchies (figure 6.42). They used the simulator to compare a direct genotype encoding of the morphology and neural networks with one based on Lindenmayer systems. Previously, Linden-

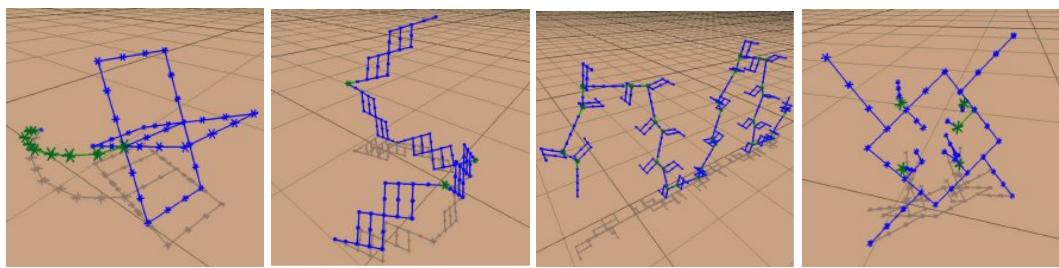


Figure 6.42: Various creatures known as “Genobots” created by a Lindenmayer based development system. Note the symmetry and feature repetition reminiscent of biology. Credit for image: Greg Hornby and Jordon Pollack [200, 204, 205]

mayer developmental encoding systems had been used to evolve morphological structure (section 6.13), but this was the first time they had been used for development of both morphology and control.

The dynamical control system consisted of individual oscillators connected to each joint. The oscillation frequency and phase offset were evolved, and there was no centralised control or communication between oscillators, and no sensory feedback.

Hornby and Pollack’s work showed that Lindenmayer systems could successfully be used to evolve control and morphology of virtual creatures. The set of rewriting rules (the grammar) was artificially evolved. Construction of the morphology and control system was by interpreting the final string as a sequence of instructions to be carried out by a 3D “turtle”. The instruction set incorporated a set of registers representing various morphological features, and an operating stack that allowed the current context of the turtle to be pushed and popped, so that branches could form in the developmental process.

The same year (2001) Hornby reported that they had successfully evolved a creature with a Lindenmayer based genotype and transferred it to reality [201]. The parts that the creature was constructed from constrain the movement to be two-dimensional, so that the path of the creature is a straight line (figure 6.43). The details of the genotype representation and dimensionality preserving mutation operators are contained in a later research summary [261].

A more detailed description of the instruction set and development process, along with some analysis of the benefits of the Lindenmayer encoding versus a direct encoding were published in 2002 [202]. Hornby showed that mutations in Lindenmayer encoded genotypes generally had a larger effect on the phenotype than mutations in directly encoded genotypes, and that these effects were more likely to be beneficial. The

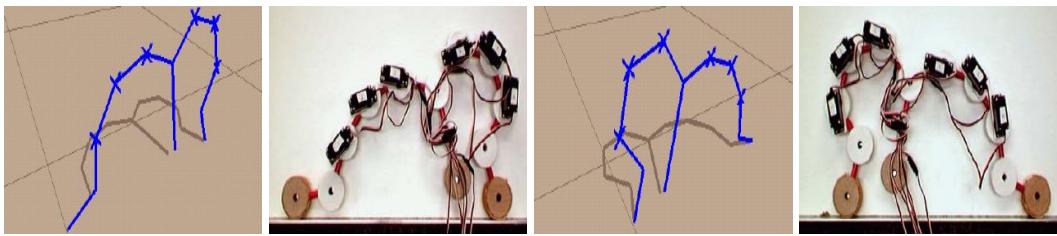


Figure 6.43: Two-dimensional creatures, known as “Tinkerbots”, move along a straight line in a virtual environment. The optimised designs are transferred to the real world by construction from modular motorised blocks.

Credit for image: Greg Hornby, Hod Lipson and Jordon Pollack [204, 341]

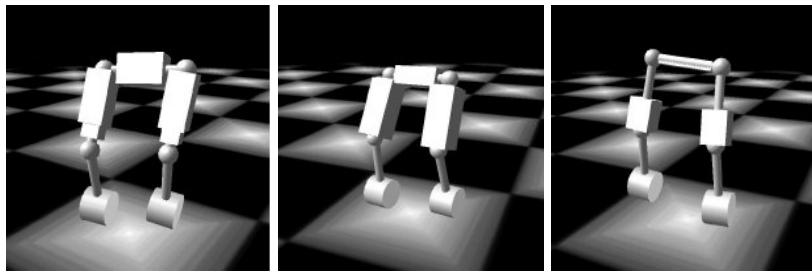


Figure 6.44: Neural controllers and weight distributions were evolved for these biped walkers. Note how the evolved weight distribution changes the appearance of the morphology.

Credit for image: Chandana Paul and Josh Bongard [328]

conclusion was that the developmental encoding reshapes and structures the search space, biasing the search to areas that contain features reminiscent of good solutions.

The co-evolution of morphology and control can be used to optimise attributes of a morphology within some constraints, e.g. whilst maintaining a desired shape. In 2001 Paul and Bongard showed that the mass distribution of a set of biped legs, including hips and a body connecting them, could be simultaneously evolved along with the neural control system [328]. Biped walking is still considered a difficult task, and this was the first time that co-evolution had been used in solving it, even though the morphology was not evolved from scratch. The genotype directly encoded the neural weights and mass of each body part. It was shown that an evolutionary process in which small changes were made at mutation time produced fitter individuals than one with larger changes. Figure 6.44 shows some of the evolved walkers.

In 2002 Endo published very similar work, also optimising characteristics of a pre-defined biped morphology whilst simultaneously evolving a walking controller from scratch [123, 124]. In this case, limb lengths, hip lengths, and servo geometries were

optimised. The parameters for the initial design were taken from a real humanoid robot which the same team had previously built and released as an open source design [485]. Distance travelled and energy efficiency were both used as fitness functions. Simulated biped walking was successfully evolved using networks of sigmoid neurons, and networks of oscillating functions.

In 2003 Shim presented his work on evolving flying creatures (figure 6.45) [390] . The morphology consisted of a tree structure with the main body being the root, and wing segments as children. Body length, wing position, angular range of movement, and wing lengths were evolved. Each wing segment was rigid, with simulation being carried out by the *Open Dynamics Engine* physics engine. The forces generated by wing flapping were approximated by a function based on wing velocity and area. Proprioception sensors provided feedback of joint angles and gyroscope levels to the neural network. Control networks, responsible for generating flapping wing motion, consisted of nodes that generated sine, cosine, and saw-tooth waveforms, and others that performed general arithmetic.

In 2004 Shim demonstrated evolution of underwater creatures and path-following behaviour (figure 6.45) [392, 393]. The creatures were constructed from capped cylinders, and fluid dynamics equations for resistance and drag of capped cylinder geometries were used to generate forces which were applied directly to the simulated dynamics bodies. To generate path following behaviour a fitness function was used which calculated the squared sum of deviations of body position from a pre-determined path in 3D space. Hence, unlike many previous neural networks in research, which only generated simple motor oscillations repeated on a scale of seconds, Shim's controllers were forced to evolve dynamics that not only controlled oscillating movements over a scale of seconds, but also had to orchestrate precise changes in these movements over several minutes in order to effect shifts in trajectory.

In 2006 Shim extended his earlier work on flying creatures, using a parameterised genetic algorithm to evolve various species differentiated by range of allowable mass, and then analysed the evolved morphologies and stability of the neural control systems [391]. The evolved creatures of different mass displayed different strategies for flight, whilst creatures with similar mass had similar strategies. The evolved creatures displayed similarities in appearance and motion with biological species of similar mass, suggesting that the evolutionary process had managed to capture the essential trade-offs between mass, size, and maneuverability.

In 2003 Vaughan evolved passive biped walking (figure 6.46) [463]. Similar to Paul

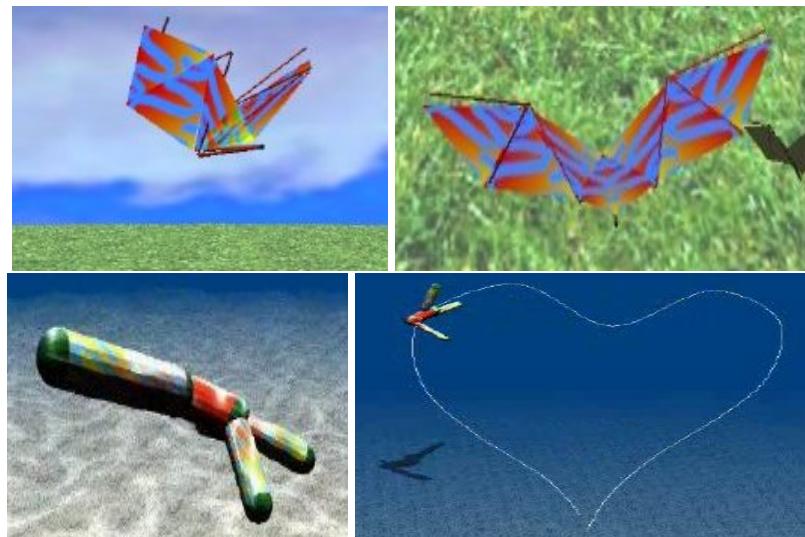


Figure 6.45: *Creatures evolved for flying, swimming, and 3D path following.*

Credit for image: Yoon-Sik Shim [390]

and Bongard's work in 2001, the basic structure of a biped was fixed, whilst morphological parameters such as limb lengths and weight distribution were evolved alongside a neural network controller. The initial evolution of a downhill passive walker was considered to be a more viable strategy than evolution of powered walking. Staged evolution could then be used to later add power. An analogy was drawn with the Wright brothers and initial developments in human understanding of flight, where designers first used unpowered gliders projected from a height in order to analyse and adapt the dynamics of aircraft as a prerequisite towards powered flight.

The neural network topology was fixed, weights and time constants were evolved. The initial task of the neural network was simply to initiate passive downhill walking. After evolving this, powered downhill walking was evolved with the neural network driving ankle motors. At this point sensory input was added to the network, beginning with low weight values in order to minimise disruption. The surface of the ground was slowly raised through many generations until it was level. This resulted in successful powered walking across a flat surface. They also managed to evolve dynamic mechanisms for adapting to the constant noise typically experienced by a semi-disabled body.

In 2004 Vaughan showed that proprioceptive feedback of only hip, knee, and ankle velocities into a feed forward network with a single hidden layer, could generate accelerations and rotations of hips, knees, and ankles sufficient for walking [461]. This was significant, as it showed that central pattern generators formed by recurrent networks are not necessary for the complex biped walking task. It was also shown that robust-

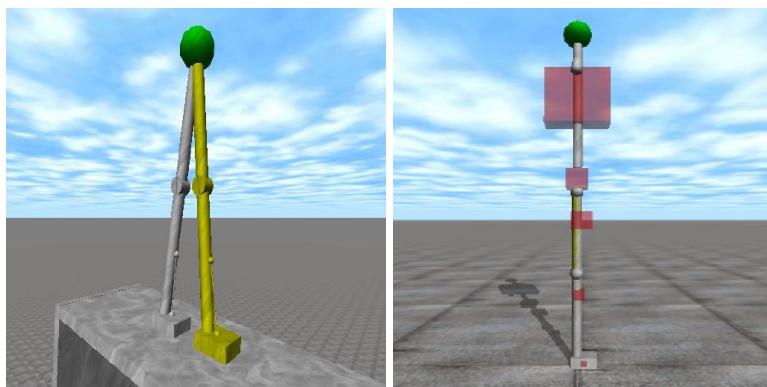


Figure 6.46: *Staged evolution was used to evolve passive downhill, and then powered, biped walking. The image on the right shows a walker with distinct head and body parts; the volume of the boxes illustrates the distribution of body mass.*

Credit for image: Eric D. Vaughan [462, 463]

ness to external forces and internal phenotype developmental errors could be evolved by varying those factors during fitness evaluations, proving that an otherwise identical control system could adapt to different bodies based only on the altered sensory feedback.

The body was extended by adding a torso and head, and then evolving for the ability to carry increasing weights [462]. A different method was used to power walking, in which a constant force is applied to the robot during simulation, pushing it either forwards or backwards. The leg dynamics and control system evolved to turn this force into stable walking motion. The evolutionary pressure of increasing the weight to be carried demonstrated the efficiency of the evolved walker; for a 200% increase in total body weight, walking required only 51% more energy. A summary of the work was published later [180].

In 2004 Giuly reimplemented the work of Sims, and added a Lindenmayer system genotype encoding [158]. The software is available as an open source tool chain, making this the first freely available system for evolution of morphology and control.

In 2004 Bongard evolved a robot that itself contained an internal model of its body which it could use to predict the outcome of its actions. To do this he co-evolved both a morphology estimate of the real morphology, and a neural control system (figure 6.47) in what he termed an “inverse evolutionary algorithm” [33, 35, 263]. Unlike the other research presented in this section, the actual morphology here was fixed, what was evolved was an estimate of the weight distribution of the actual morphology. This estimated morphology was then used in fitness evaluations of the neural control system,

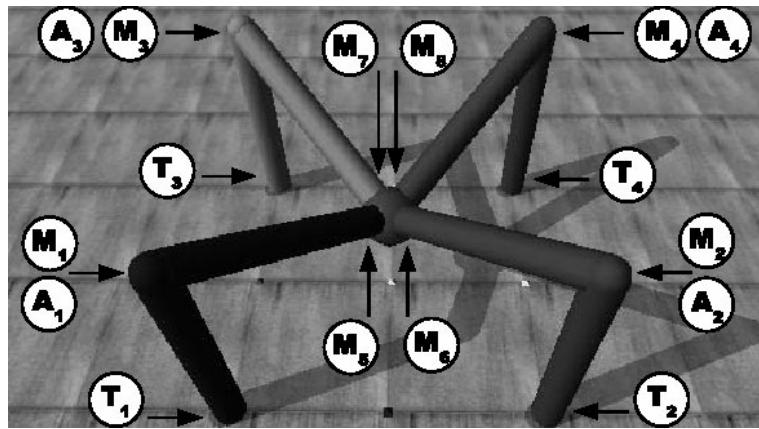


Figure 6.47: This robot evolved an internal model of its own body which was then used to evolve an adaptive controller. Both models are updated in real-time to compensate for damage.

Credit for image: Josh Bongard [35, 263]

which directly compared actual sensory readings to expected ones generated by the internal model, and selected for the most accurate models.

Thus there was a disconnect between the evolution of the neural control, performed on an estimate of the morphology, and the actual morphology. It was shown that the estimate correctly converged towards the actual physical characteristics of the robot. Using an estimated model of the real morphology has the advantage of allowing the evolutionary process to adapt to variability in the morphology. It was shown that the control system could adapt to various types and severity of physical damage.

In 2005 Miconi and Channon reimplemented Sims's work, but using a standard neural model rather than wave generators, and with some other minor differences [295, 297, 298]. They claimed that this was the first complete reimplementation of Sims's work, discounting the Taylor and Massey reimplementation in 2000 as being “incomplete”, and discounting Ray's reimplementation because evolution was driven by aesthetics rather than locomotion fitness tasks [295]. The major difference between this implementation and Sims's was the use of sigmoid neurons rather than waveform generators to control the virtual creatures. The system was used to evolve creatures that displayed locomoting and box grabbing behaviours. The source code for this implementation was released as open source.

In 2005 Ventrella evolved 2D agents known as “Swimbots” [466]. Unlike most of the research in this section, this work used a virtual life system, with agents living in the same world, gaining energy from consuming food, and mating to produce offspring.

Ventrella showed that there was a delicate balance between morphological function and sexual attractiveness. Agent attraction was evolved for various parameters such as colour and size, but agent morphologies were constrained by the requirements of gathering food and reproducing.

In 2006 Chaumont reported a reimplementation of Sims's work and used it to evolve virtual catapults [60, 61]. The genotype was constrained to have a small block initially fixed to the evolved creature. Some time after the start of the simulation this joint was destroyed, leaving the block free to travel away from the creature. The fitness function measured the distance between the creature and the block after a few seconds had elapsed. Thus evolutionary pressure was to evolve an agent that produced some rapid motion to accelerate the block to a high velocity coinciding with the destruction of the connecting joint.

In 2007 Lassabe *et al.* reimplemented Sims's work, but using evolved waveform generators with a classifier system to select some waveform given a pattern of input stimulus [255]. A randomly generated global table of 1000 patterns was initialised. The genome consisted of the morphology of a creature and a set of mappings from specific input stimulus to sequences of indices into the pattern table. When a particular stimulus was classified the corresponding pattern sequence would be composed into a single waveform and set to a joint motor. This is an interesting approach since it allows composition of novel waveforms, and sensory inputs can be used to drive different behaviours. The evolved behaviours were locomotion across uneven surfaces and up steps, and cooperative block pushing.

In 2008 Miconi evolved virtual creatures in an artificial life world [296]. The creatures used the same physics morphologies and controllers as his previous 2005 work (which had reimplemented Sims's "Blockies"). Creatures were able to gather energy, injure each other, and reproduce by cloning. Running out of energy resulted in death. The virtual world was a large sphere known as the "Evosphere" (figure 6.48).

6.15 Evolving modular robots

Modular robotics follows the concept that complete agent morphologies can be assembled from generic modules that are capable of performing a variety of actions, but will be specialised when placed into specific positions in the morphology. The first co-evolution of a modular robotic based morphology and control system was carried out by Marbach in 2004 (figure 6.49) [276]. Each module is a cube, and has a stan-

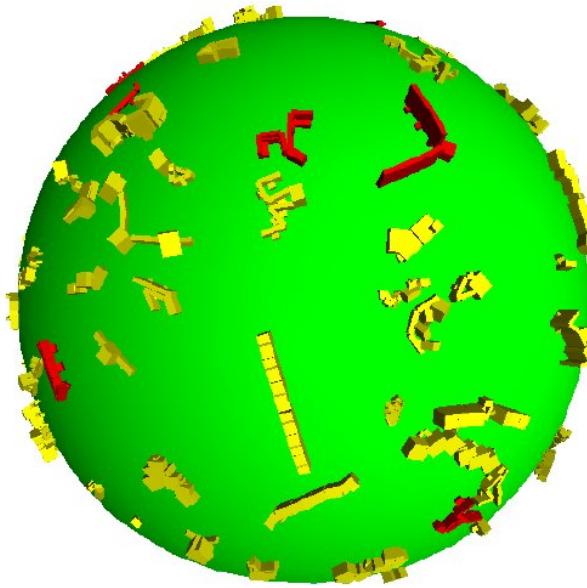


Figure 6.48: 3D virtual creatures evolved in an artificial life world. Each creature has an energy level which can be increased by gathering food, and which is decreased through activities such as moving, reproducing and fighting, which have associated energy costs.

Credit for image: Thomas Miconi [296]

dardised joint and flat connection surface, enabling any number of identical modules to be connected together to form a complex composite robot.

In 2005 von Haller presented his work on evolved underwater modular robotics derived from Marbach's system, known as "Neubots" (figure 6.49) [468, 469]. Each module is a cube, with a magnetic connector and hinge joint on each face, producing 6 degrees of joint freedom per module. Movement of each module is produced by eight water jets, resulting in 4 degree of freedom movement. Neural oscillators were evolved first, with the frequency being controlled by an input signal. Then morphology and control of single composite robots built from many interacting modules was evolved for the locomotion task. It was shown that composite underwater robots could be successfully evolved, and that the input signal to the oscillator did indeed cause the creature to speed up or slow down as hypothesised.

6.16 Summary

This chapter has described past research where solution genotypes have been evolved using genetic algorithms. There are two basic approaches to mapping a problem and

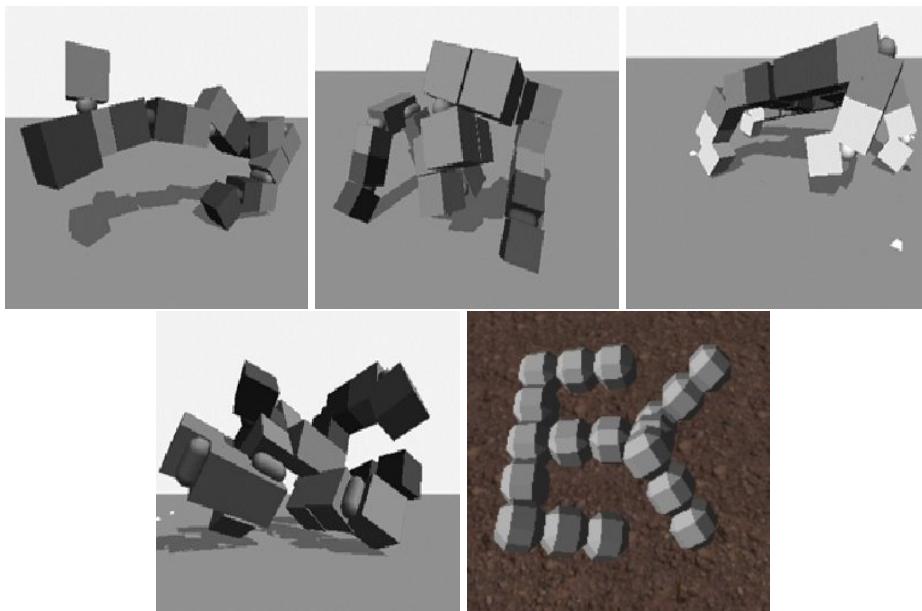


Figure 6.49: These modular robots are assembled from homogeneous blocks. Together, they act as a composite unit displaying emergent locomotion. The bottom right image is of a “Neubot” underwater swimmer.

Credit for image: Daniel Marbach and Bartelemy von Haller [276, 468, 469]

its solutions onto a genome. One is to use a binary string genome, map the parameters of the problem onto bit strings, and concatenate them to produce the full genome. Generic binary string reproduction operators can be used to crossover and mutate the genome without requiring knowledge of the problem domain. The other approach is to use a problem specific genome, usually some form of directed graph, and to devise reproduction operators that are applied directly to this abstract structure. It has been argued that the binary string approach is a more faithful analogy to biological reproduction, and that it avoids inadvertently biasing the search, but it has also been argued that higher level abstractions enable more powerful operators to be used, which can positively bias the search and hence accelerate evolution.

Many different types of artificial evolution have been presented. The field of artificial evolution is huge, so the examples given have focussed on specific sub-fields that are relevant to this thesis. The idea that evolved individuals should show some *emergent* properties is important to many of the areas surveyed; it is not enough that individual nodes of a neural network can individually sense and control specific bits of a body, they must communicate and work with other nodes in order to display an orchestrated and coordinated global behaviour.

Methods and results of evolving different types of neural networks (continuous, spiking, reduced) and other types of continuous and discrete network (e.g. electronic circuits) have been presented, including some targeted at robot control applications. This thesis will compare the performance of some continuous and discrete network controllers in robot control applications, and previous research suggests that problem specific encodings may accelerate evolution, and that use of a developmental encoding will result in individuals that display more reuse of components, more symmetry and higher fitness.

Developmental encodings have been highly successful in nature, enabling, for example, mammal embryos to grow from a single cell to an advanced multicellular organism with over 10 trillion cells, whilst at the same time the developmental process is robust in the face of a varied environment. Each embryo experiences a unique environment in the womb, with slightly different levels of various molecular species, and individual cells must be robust to both intra-cell and inter-cell variances. Despite this, the process is robust within a regular environment, with major developmental errors occurring infrequently. It has even been shown that the process is robust to some major changes in environment, such a lack of gravity resulting in weightlessness; experiments on weightless embryogenic development have resulted in viable offspring for several species, including one mammal (mice) [473].

Both continuous and spiking neural networks have been evolved to control 3D robots. Some previous works are particularly relevant to this thesis. One is that of Floreano, who evolved neural networks with 8-bit integer node dynamics and spiking communications to control a wheeled robot carrying out a wall avoidance task. This was the equivalent of a 256-state quantised model. Continuous versions of the same controller failed to evolve, although this was almost certainly due to the limited genome that did not allow varying connection weights, regardless no comparison was possible.

Sims's work on evolving 3D creatures and network controllers has been influential in the field, and is directly relevant to this thesis, as is the work of all who followed him in evolving both the morphology and network controllers of whole creatures. The evolution of pattern generators has a direct application to the evolution of locomoting creatures, as pattern generators are necessary to drive cyclic motions that in turn drive muscle and motor actuators. The successful evolution of modular robots, and the success of modular composition in biological life, particularly when viewed at the cellular level, suggest that modular robots, and a genome that allows reuse of components, is beneficial to evolutionary success.

One notable point is that the bulk of work done so far on evolutionary robotics has assumed that continuous neural models are necessary. Given that many target applications of these controllers involve mobile robotics platforms, running from battery power, it is perhaps surprising that little research has been done to evolve simpler models, or to compare and contrast the performance of models which vary in their computational requirements.

The common use of PCs for neural network modelling provides a platform capable of simulating loosely connected networks of thousands of neurons in real-time with floating-point arithmetic. This will consume a large amount of power (typically over 150 Watts for sustained operation), which in turn will dissipate a large amount of heat. The computational resources available on the latest 64-bit multi-core PC processors, which possess integrated 128-bit SSE vector processing units, are also much greater than those in typical embedded robotics platforms, which often rely on low-power CPUs capable only of integer arithmetic.

The availability of relatively powerful PCs, connected to mains power supplies, and their ease of use, has perhaps led many researchers to overlook the idea of evolving alternative neural control systems that are based on digital logic and optimised for low power and low computational resource environments. Similarly, the use of mains powered PCs based on the von Neumann architecture, with large DRAM based memories for storing neural networks, has provided little incentive to experiment with neural control systems that use non-synchronous timing models, since they would only substantially reduce power consumption or enable greater scalability when deployed on alternative, perhaps custom built, architectures.

Chapter 7

Overview so far

This thesis investigates whether digital, quantised neural models utilising reduced precision arithmetic can be used for tasks where evolved floating-point neural networks are currently dominant. The experiments that will be described in the following chapters are the evolution of a neural network to control a pole balancing robot (which is a traditional AI control problem), and the combined evolution of a complete robot morphology and control system. These tasks have been chosen as they represent common tasks pursued by previous researchers. The evolution of a combined morphology and control system was desired as it has been claimed previously by other researchers that this allows a synergistic evolution to take place, not only by making the evolution itself more likely to be successful, but also by making the evolved controllers more likely to be of a higher fitness. It also represents a more complex and interesting problem than pole balancing.

The aim of these experiments is to see if it is possible for reduced neural networks to control a robot, and if so, to quantify the change in performance between models with varying levels of complexity. In order to do this, the type of neural network to be evolved in different experiment replicates will be varied between different implementations of some biological models (sigmoid, spike response model, integrate-and-fire, Beer's continuous time recurrent neuron, models from Taga and Ekeberg). Quantised versions of each model will be implemented, so that the effect of changing the quantisation (and arithmetic precision) of each model can be tested experimentally.

In order to carry out the experiments, custom software is required. The software must define some schema to represent neural networks, and include a simulator to simulate the operation of a given network. Both of the experimental setups will be performed within a simulated 3D world, and this will require use of a physics simu-

lator to calculate the location and motion of objects within the world, and also a 3D visualisation program to enable the user to observe the simulations. Software will be required to perform the process of a genetic algorithm, which represents the genome being evolved using some schema, creatures various populations, calls an evaluation function to test the fitness of the genotypes, and performs reproduction to generate the subsequent populations based on the fitness values assigned to the current one.

Chapter 8

Software design

The previous chapters have introduced genetic algorithms, and their use in the evolution of various types of neural networks and 3D morphologies. Successfully carrying out such experiments requires the development of a large amount of custom software. This chapter describes the functionality of the software developed for this thesis, including genotype codings and phenotype simulators, and justifies various design decisions made along the way.

8.1 Creature morphology

The robot morphologies were to be evolved using a genetic algorithm. This meant that a robot phenotype had to be designed, in which the morphology and control system would be constructed from a suitable set of building blocks. These blocks had to be connectable, tractably simulatable, and versatile enough to offer the chance of evolving sophisticated morphologies. Hypothetically, any kind of geometric primitive which allows size to be varied should be able to meet these requirements. Sims used a cuboid geometric primitive to construct his *Blockies* and produced a range of diverse creatures [397]. Komosinski used spheres connected by springy lines for the creatures in his *Framsticks* world [243]. Taylor and Massey used cylindrical primitives to reproduce Sims's work [436].

After some consideration it was decided to use capped cylinders (figure 8.1) as the basic geometric building block. These geometry objects consist of a cylinder with hemisphere caps on both ends. This has several advantages. A joint joining two connected cylinders is placed in the centre of the two connected end caps, so if desired it would be possible to implement collision detection between the two body parts by

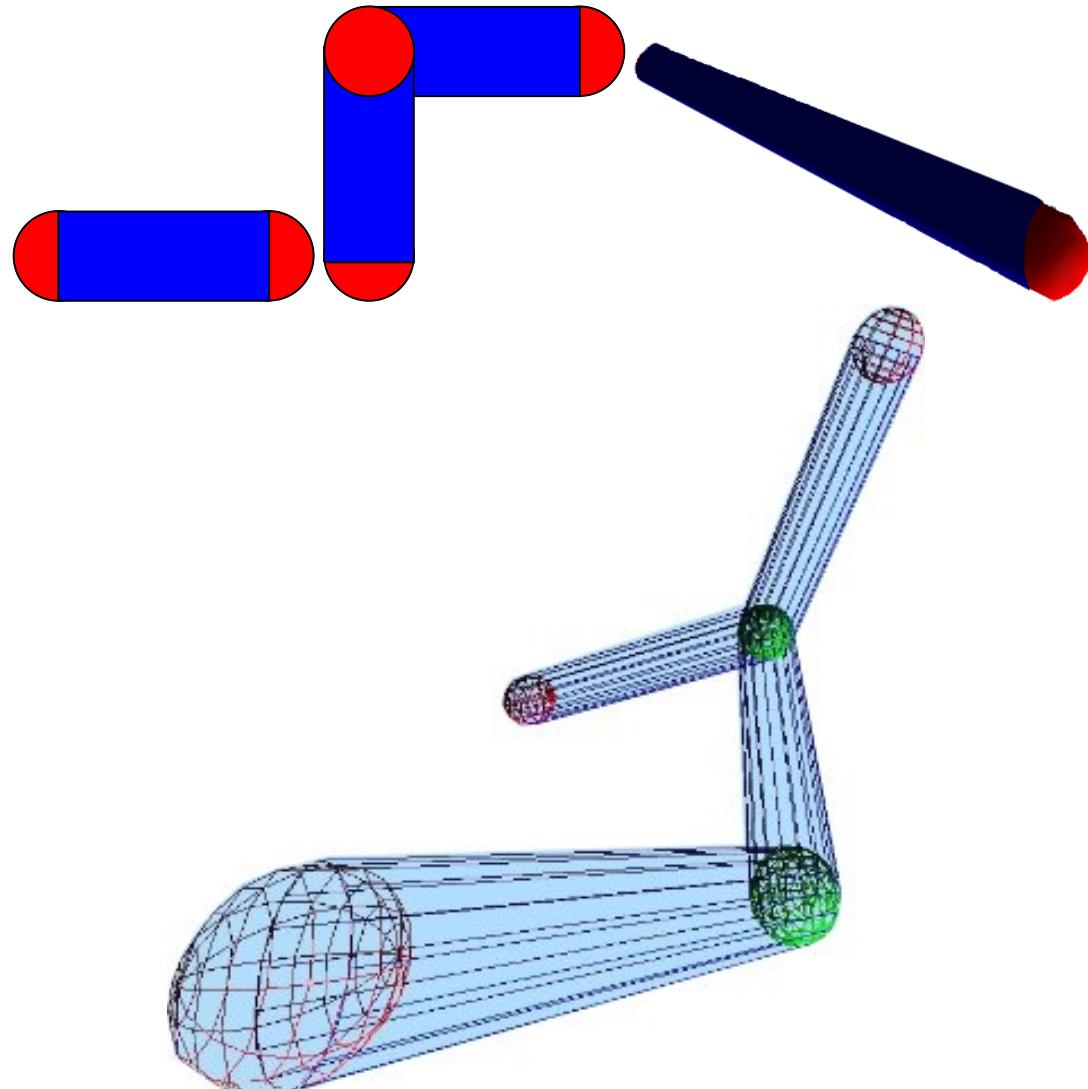


Figure 8.1: A “capped cylinder” is the basic geometric primitive from which morphologies are constructed. It is a three-dimensional cylinder geometry with hemisphere caps at the end. Cylinders are connected together to construct multi-cylinder morphologies. The centre point for these joints is the centre of the hypothetical sphere that would be described were the hemisphere caps to be rotated. To eliminate flicker when animated, these hemisphere caps are simply rendered as a single sphere. Top from left: 2D side view of the capped cylinder primitive, two capped cylinders jointed together, and a solid 3D render (with perspective and shading) of a capped cylinder. Bottom: A wireframe 3D render of a complete four cylinder morphology.

calculating collisions between the cylinders and ignoring the permanently interpenetrating hemispheres (this idea, whilst more biologically faithful, was later abandoned after it was discovered that it too heavily constrained the morphology and trial evolutionary runs failed to produce any working creatures). The end caps are hemispheres which means that their rotation through and around the same point describes a sphere. This sphere can be rendered instead of rendering the, possibly many, hemisphere end caps. A single sphere has the advantage of always providing smooth and consistent animation at the joint. The use of hemisphere end caps provides for more realistic curved contacts with the ground plane, as most animals have smooth, curved digits rather than the sharp angles reminiscent of cuboids and some other geometric primitives.

Each capped cylinder has parameters and attributes which define its location in 3D space, its rotation, its length, and details of the joint connecting it to the rest of the robot. These parameters are configured from data in the genotype during morphogenesis.

8.2 Morphogenesis

Morphogenesis is the process of converting the genotype into a system of geometric bodies, joints, and constraints that can be simulated using the *Open Dynamics Engine* physics simulator. Figure 8.2 shows some example mappings from genotype to phenotype. The genome is a directed graph, which possibly contains cycles, and the phenotype is a tree of capped cylinders. The cylinders in the phenotype are connected by joints with either 1, 2 or 3 degrees of freedom.

The simulator does not allow geometry cycles to occur, as the constraints are unsolvable, so the genotype graph has to be unrolled and converted into a tree structure before it can be simulated. The unrolling process is simple, and was inspired by that used by Sims [397, 398]. Starting at a root node (which is an evolvable parameter), the process will conduct a “depth-first search” to find paths through the graph. Each visited node is used as a genotype template to create a capped cylinder in the phenotype, with the edges of the genotype graph directing how the phenotype cylinders are to be connected. In order to prevent infinite loops when faced with cycles in the genotype graph, each genotype node specifies a *recursion limit*, which is the maximum number of times that the node can be visited in any particular walk of the graph. The *recursion limit* is an upper bound on the number of times that a node can be visited during a single walk, not on the total number of instantiations of a particular node in the phe-

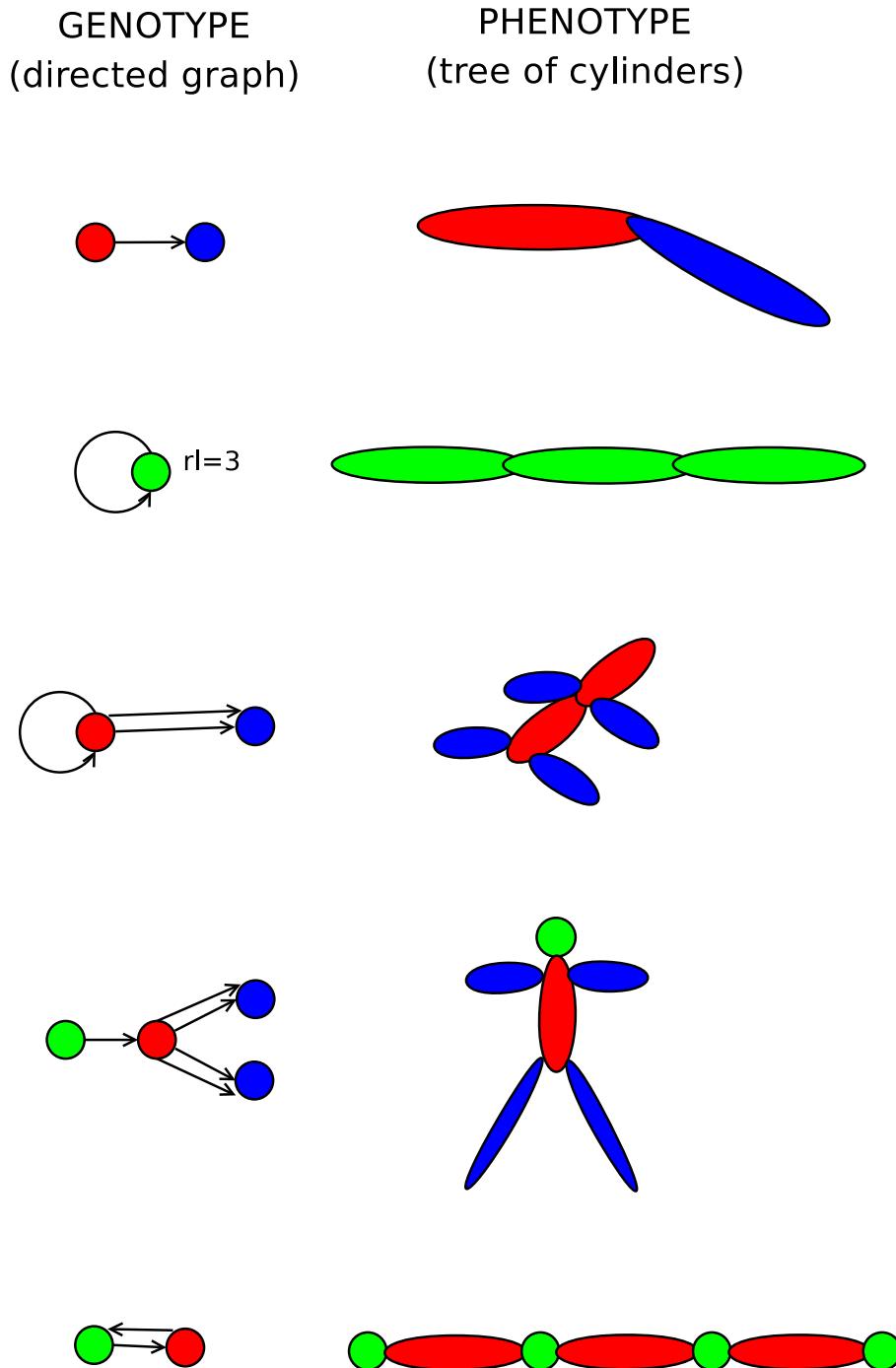


Figure 8.2: *Example genotype to phenotype mappings.* A directed graph is unrolled by conducting a walk of the graph, starting from the root node, and using every visited node as a template to create a hierarchical tree of capped cylinders. A recursion limit (specified as $rl = 3$ for the second example, unspecified for the others) restricts the number of times that a particular node can be mapped onto a cylinder in any particular walk of the graph.

notype, so varying paths that visit the same genotype node will cause the node to be instantiated more than once in the phenotype. A depth-first walk is terminated when it either reaches a node with no outgoing edges, or where the edges only go to previously visited nodes which have already reached their recursion limit. At this point, the node will be considered a leaf node and the search process will back-track to the parent and carry on just like a normal depth-first search.

8.3 Evolution of the morphology

Each cylindrical body part has some evolvable parameters which affect where it is placed, how it is connected to its parent cylinder, and the way in which it is simulated. These parameters are part of the genome — each node of the directed graph genotype has its own copy of these parameters. The default mutation rate was set to 5% based on pilot runs; this means that the probability of any given node, edge, or parameter being mutated during genotype reproduction was 5%. The parameters and their valid value domains are:

Parameter	Valid values	Description
scale	(0.2, 5.0]	Cylinder length relative to parent
recursion_limit	{0, 1, 2}	Upper limit on node use when unrolling
joint	{hinge, universal, ball}	1, 2, or 3 DOF joint
axis1	(x,y,0)	Axis of joint — unit vector on the x, y plane
rotation	Quaternion	Rotation relative to parent
lostop	$\{-\infty, (-\pi, 0]\}$	Lower limit of first axis on joint
lostop2	$(-\pi/2, 0]$	Lower limit of second axis on joint
lostop3	$\{-\infty, (-\pi, 0]\}$	Lower limit of third axis on joint
histop	$\{\infty, [0, \pi)\}$	Upper limit angle of first axis on joint
histop2	$[0, \pi/2)$	Upper limit angle of second axis on joint
histop3	$\{\infty, [0, \pi)\}$	Upper limit angle of third axis on joint

A more detailed explanation of these parameters follows:

scale Figure 8.3. The scale is a specification of the length of the cylinder relative to the length of its parent cylinder. Only the root cylinder specifies an absolute length. Note that it is possible for the same genotype node to serve as a template for phenotype cylinders with different absolute lengths if the genotype node has incoming edges from two or more other nodes.

recursion limit Figure 8.4. The recursion limit specifies an upper limit on the number of times a genotype node can be instantiated as a cylinder in the phenotype during any given path of the “depth first search” through the graph.

joint Figure 8.6. Specifies the joint type. ODE has three joint types — hinge, universal, and ball, with 1, 2, and 3 degrees of freedom respectively.

axis1 Figures 8.6 and 8.7. Specifies the base axis of rotation for the joint and motor of this cylinder. This parameter is ignored for hinge joints (a hinge joint has only one valid axis because the positions of the two cylinders and the hinge itself are fixed). For a universal joint, this parameter describes the first axis that the joint will rotate around. The second axis is calculated automatically — it must be perpendicular to the first axis, so we calculate $axis1 \times (0,0,1)$. Axes are not required to be specified for a ball joint as it has 3 degrees of freedom. However, the motor model allows the user to set desired velocities around the axes independently, and so this does require specifying the axes. Again, this only requires a single vector parameter to be specified in the genotype, as the other two axes can be calculated given the position and rotation of the cylinders.

rotation Figure 8.5. Rotation is specified relative to the parent cylinder.

histop and lostop Figure 8.8. Upper and lower limits on the joint’s angular motion. There are three pairs of stop limits — one for each potential axis. Hinge joints will only use one pair of stops, universal joints will use two pairs, and ball joints will use all three pairs of stop limits.

The actual topology of the genotype morphology graph is subject to mutation by the genetic algorithm, with mutation operators randomly adding and removing edges and nodes. The connections between the morphology and the neural networks are also subject to evolution by the genetic algorithm, with both connections from sensors to neural network nodes and from neural network nodes to motors being randomly added, removed and changed.

8.4 Example morphologies

Figure 8.9 shows some example phenotype morphologies evolved using the genetic algorithm.

SCALE PARAMETER - LENGTH RELATIVE TO PARENT CYLINDER

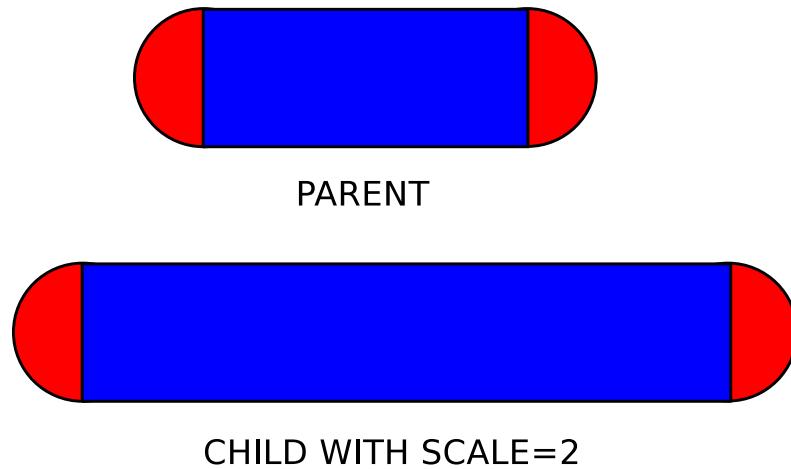


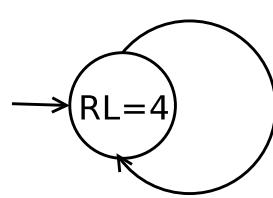
Figure 8.3: *The “scale” parameter determines length of the cylinder. In this example the scale is 2 making the child twice the length of the parent.*

8.5 Neurogenesis

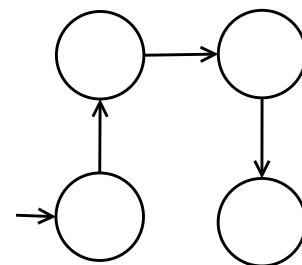
Each capped cylinder has an associated neural network (see figure 8.10) which can receive inputs from sensors indicating contact and joint angles. Each capped cylinder is only directly associated with the joint that connects it to its parent cylinder, where “parent” means the cylinder that is the direct and unique parent in the phenotype tree. The neural network of a cylinder can generate output signals that drive an angular motor on the joint between the cylinder and its parent. The joint may have 1, 2, or 3 degrees of freedom, and the motor will drive angular motion around these axes independently. The motor associated with a cylinder can only drive the joint with the parent cylinder, meaning that every cylinder apart from the root will have only one motor associated with it (the root will have no motor associated with it, as it has no parent node).

RECURSION LIMIT PARAMETER - LIMITS GENOTYPE EXPANSION

GENOTYPE GRAPH



PHENOTYPE GRAPH



INTERACTION WITH SCALE PARAMETER - PHENOTYPE CYLINDER LENGTH MULTIPLIED BY "SCALE" ON EACH ITERATION

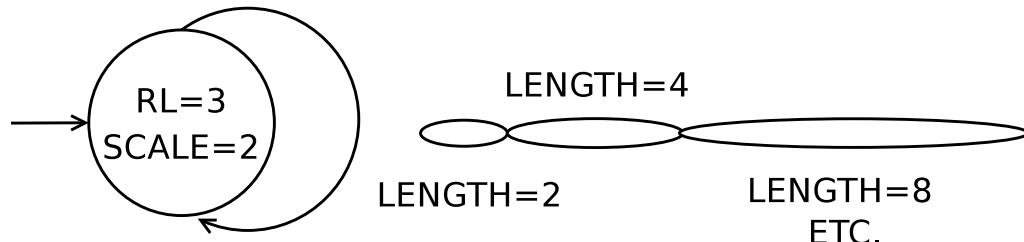
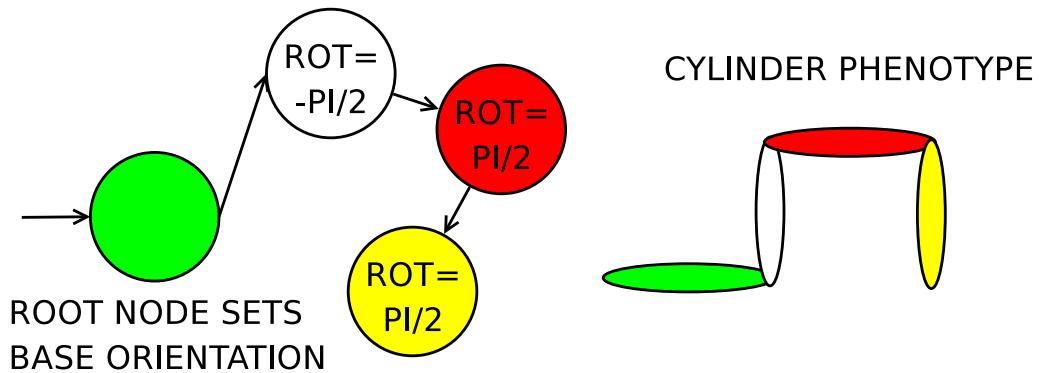


Figure 8.4: The “recursion limit” parameter specifies how many times a genotype node can be instantiated as a phenotype cylinder in a single depth first walk of the graph. It also prevents infinite loops. It can interact with the scale parameter to alter the length of child cylinders relative to the parent cylinder.

ROTATION PARAMETER - SETS POSITION RELATIVE TO PARENT



INTERACTION WITH ROTATION - PHENOTYPE
CYLINDER ORIENTATION ROTATED ON EACH
ITERATION

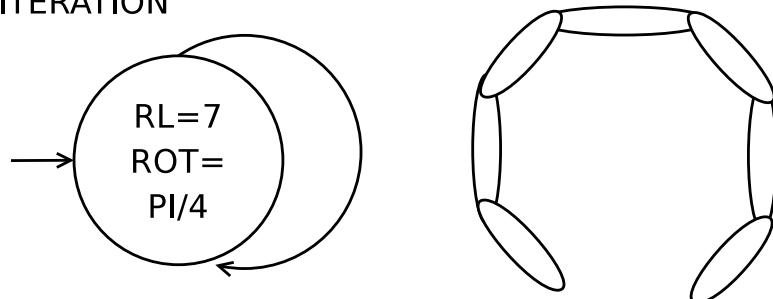


Figure 8.5: The “rotation” parameter specifies rotation of the cylinder relative to the parent cylinder. It can interact with cycles in the genotype graph (if allowed by the “recursion limits”) to create repeated sequences of similar cylinders, each rotated and scaled with respect to each their immediate parent cylinder.

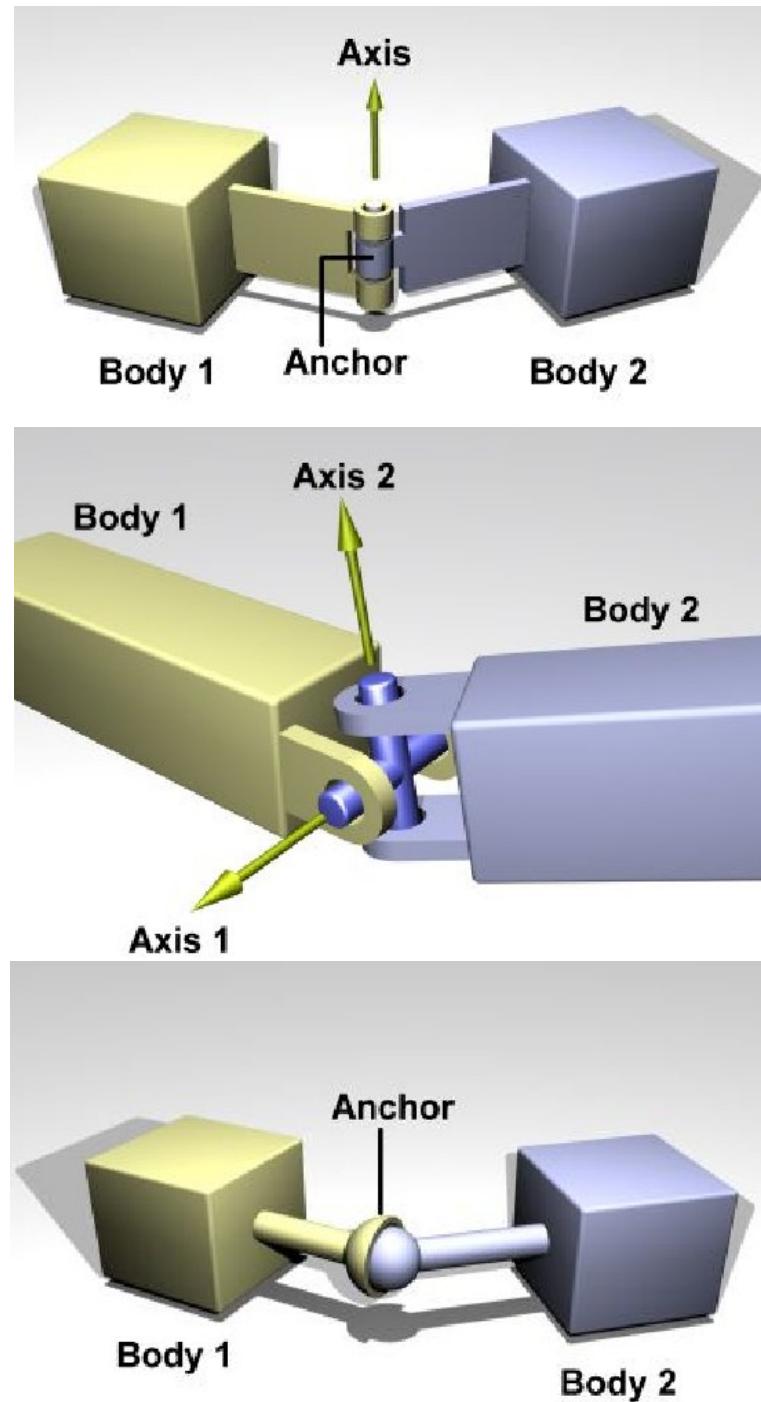


Figure 8.6: *Joint types available in the Open Dynamics Engine physics simulator. From top: hinge, universal and ball. The hinge joint allows rotation about 1 degree of freedom and requires the axis of rotation to be specified. The universal joint allows rotation about 2 degrees of freedom. It has two axis that must be perpendicular. The ball joint has no axes as it has 3 degrees of freedom and so should be able to rotate freely. However, the motor model drives rotation around axes independently and so still requires that 3 axes be specified when driving a ball joint.*

Credit for image: Russell Smith [404]

AXIS1 PARAMETER - SETS AXES FOR UNIVERSAL AND BALL JOINTS

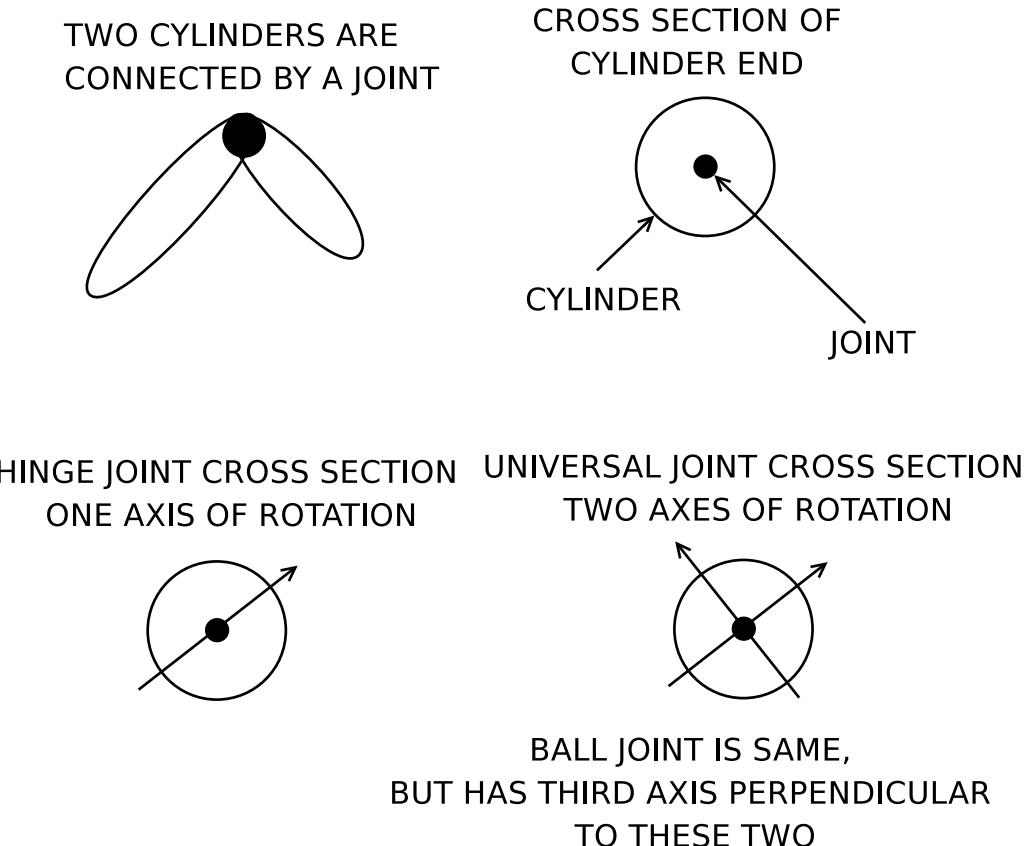
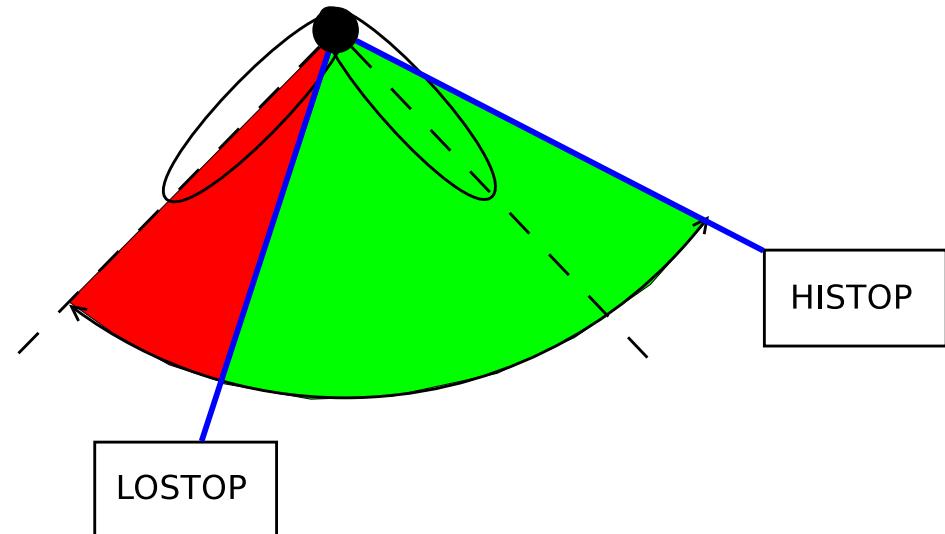


Figure 8.7: The “axis1” parameter specifies the base axis of rotation about the joint. The motor attached to the joint drives the axes independently. Only one parameter needs to be specified as any other axes must be perpendicular to this one and can be computed automatically.

LOSTOP AND HISTOP PARAMETERS - LIMIT JOINT MOVEMENT



THE HINGE JOINT ALLOWS ROTATION OF CYLINDERS RELATIVE TO EACH OTHER.

THE ANGLE IS PREVENTED FROM FALLING BELOW THE LOSTOP LIMIT OR EXCEEDING THE HISTOP LIMIT.

THE GREEN AREA SHOWS THE ALLOWED RANGE OF JOINT ROTATION.

Figure 8.8: *Stop limits constrain the joint angle to be within some specific range. Each joint has one, two or three pairs of stop limits corresponding to its degrees of freedom. This example shows two cylinders joined by a hinge, which enables them to rotate with respect to one another through one degree of freedom.*

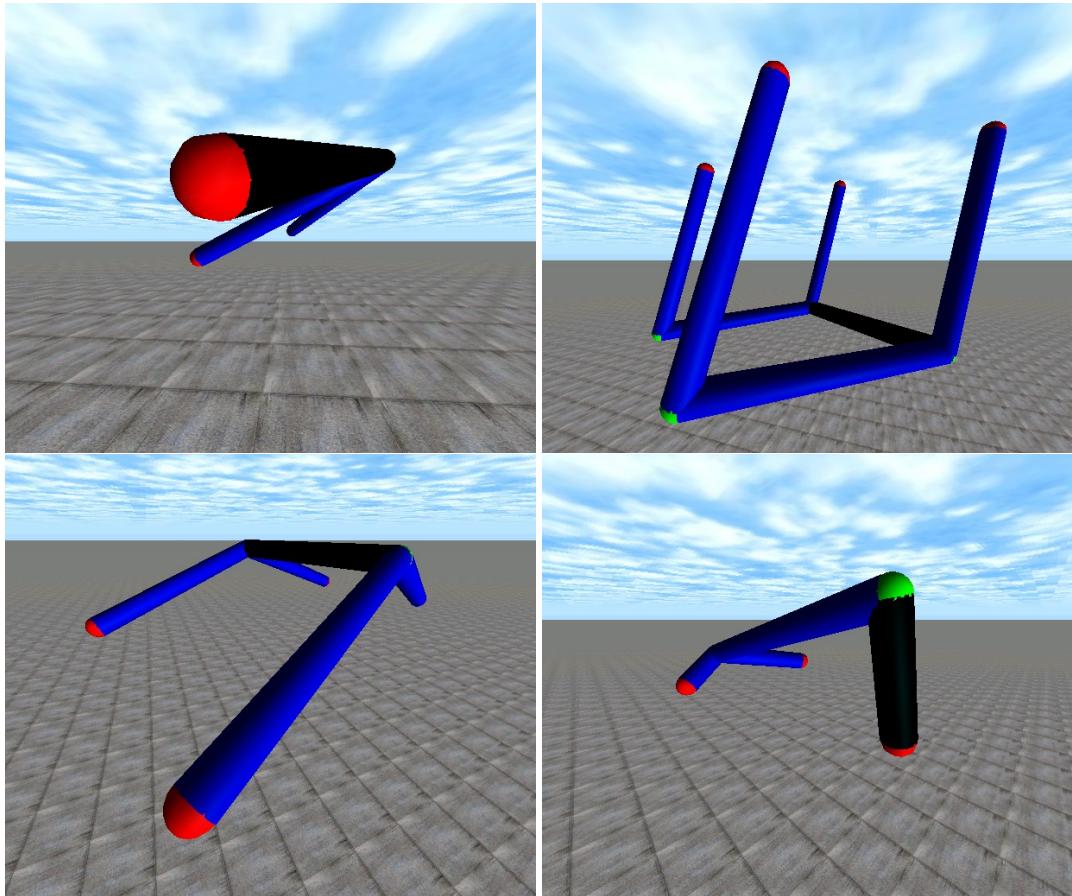


Figure 8.9: These example phenotype morphologies are intended to give the reader a flavour of the type of robot morphologies that are likely to evolve. They show repetition and symmetry reminiscent of biological creatures, indicating that the morphogenesis process is quite adaptable.

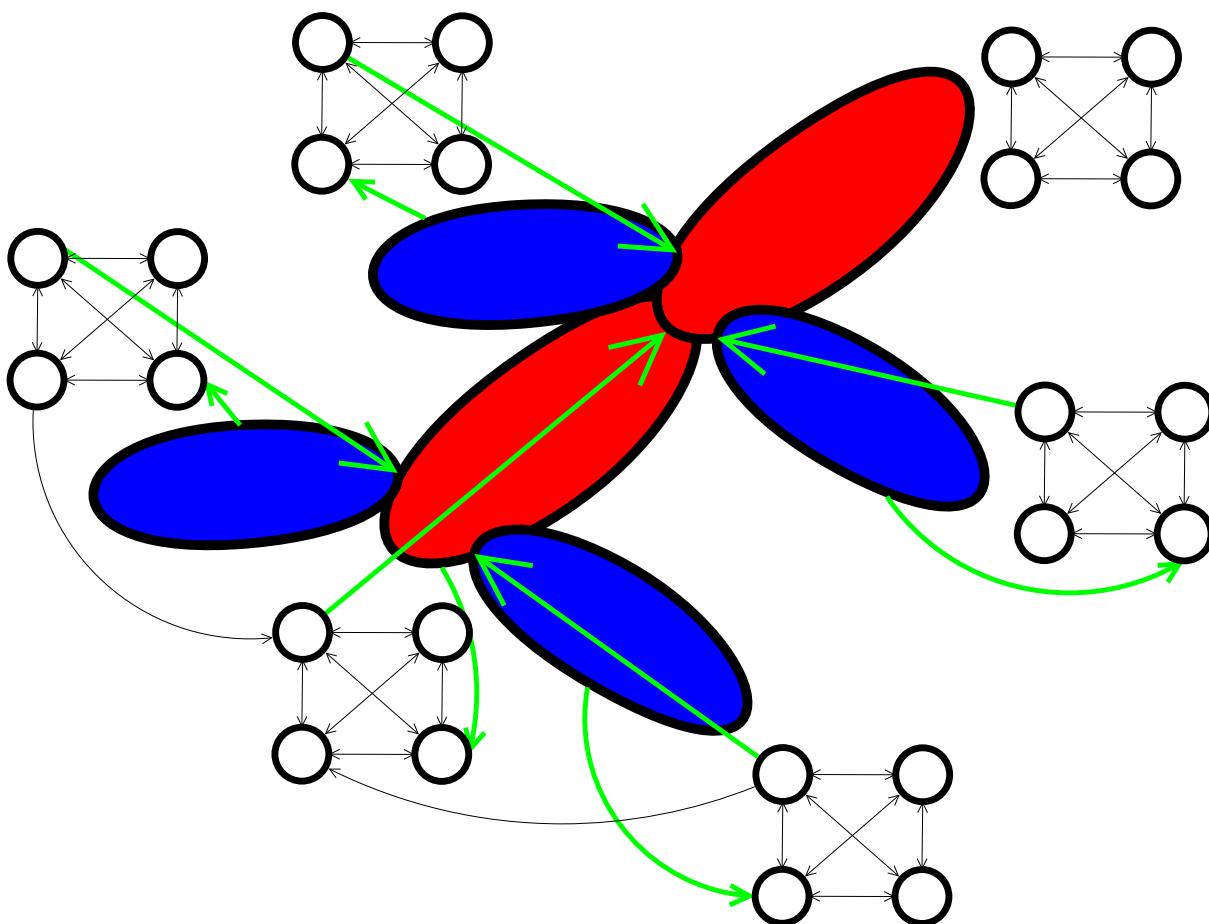


Figure 8.10: A robot phenotype showing morphology and neural control. Each capped cylinder has its own neural network, which in this example is fully connected. Each cylinder has a 1, 2 or 3 DOF joint with its parent, with a corresponding motor driven by one neural output node for each axis (green edges from neural networks to morphology). Sensors for contact or joint angle can send values to the neural network (green edges from morphology to neural networks). There may be direct connections between the neural networks of connected cylinders. The root cylinder (red, top right) has no parent, and so no motor joint or angle sensors (its neural network does exist however and could be connected to the networks of its child cylinders).

It is possible for the neural networks of neighbouring cylinders (where one is the parent or child of the other) to have a direct connection from a node of the neural network of one cylinder to a node of the neural network of the other cylinder. It is also possible for sensory and motor connections to exist between neighbouring cylinders. It was hoped that allowed communication between neighbours in this way would encourage greater global synchronisation to emerge.

Data to generate the sensory connections must be stored in the genotype. The possibility of just randomly creating connections after the graph had been unrolled was considered, but this would likely lead to unreproducible controllers. The unrolling process converts a graph into a tree, which significantly changes the topology, so it is not possible to directly store phenotype connections in the genotype. In order to generate evolvable reproducible connections in the phenotype, the unrolling process follows an iterative method of generating an unrolled tree from the current working graph, then creating one connection in this tree, and back-annotating the results to the genotype. This is done as we have no way to figure out what the eventual connections may be without constructing the phenotype, so we construct the phenotype, figure out the connections, and then back-annotate them to the genotype. Note that this is not Lamarckism; the phenotype generated during this process is never used in a simulation, and is immediately destroyed once the reproduction process is complete. Once a valid list of connections has been generated in the genotype, it is subject to evolution by the genetic algorithm, with new connections being added, and existing connections being removed or mutated.

8.6 Neural network topologies

All of the neural networks of a particular robot will have the same topology and neuron type. The terms used here to describe network geometry and connectivity are as used by Wolfram to describe the architectures of cellular automata in [480]. Each network has a possible geometrical configuration (1D, 2D, 3D, or none) which defines how nodes within a size n neighbourhood will be connected together. A neighbourhood is the set of nodes that influence the update of a given node. For example, if a network has a size 5 neighbourhood, this means that each node has 4 inputs, so its update is influenced by 5 nodes including itself. The degree of connectivity of a network is synonymous with the neighbourhood size.

In a 2D geometry the nodes are placed into a fixed 2D grid. The neighbourhood

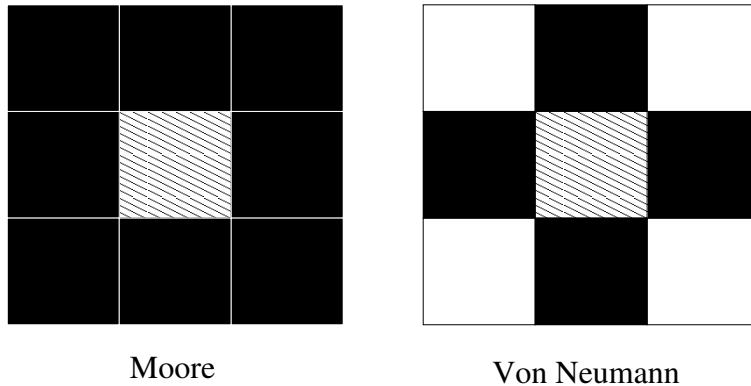


Figure 8.11: *Moore topology versus von Neumann topology*. The neighbourhood size is 9 for the Moore topology, and 5 for the von Neumann topology. The centre cell is patterned, the neighbour cells that receive connections from the centre cell are black, and the non-neighbour cells (those that are not connected to the centre cell) are white. The Moore neighbourhood has a characteristic square shape. The von Neumann neighbourhood has a characteristic cross shape. Both of these topologies can be easily extended to 3D.

topology can be “Moore” or “von Neumann” (terms from in [480]). Moore neighbourhoods are square whilst von Neumann neighbourhoods are cross shaped (figure 8.11). The concepts of von Neumann and Moore neighbourhoods can be extended to 3D by simply repeated the pattern of connectivity in the third dimension.

Figure 8.12 shows some example neighbourhoods for given network geometries. Note that this figure only shows the incoming connectivity for the single neuron marked with X — this connectivity would be repeated for every neuron, but plotting this on the graph would make the example unreadable.

The network does not have to use a geometric layout. In the absence of a geometric configuration, the nodes can be fully connected or randomly connected. A fully connected topology can be used to effectively simulate any other topology by setting the connection weights to zero. Fully connected networks do not tend to scale well, as the number of edges and their parameters increases factorially. A randomly connected network allows any pair of nodes to be connected together and hence has no underlying geometry.

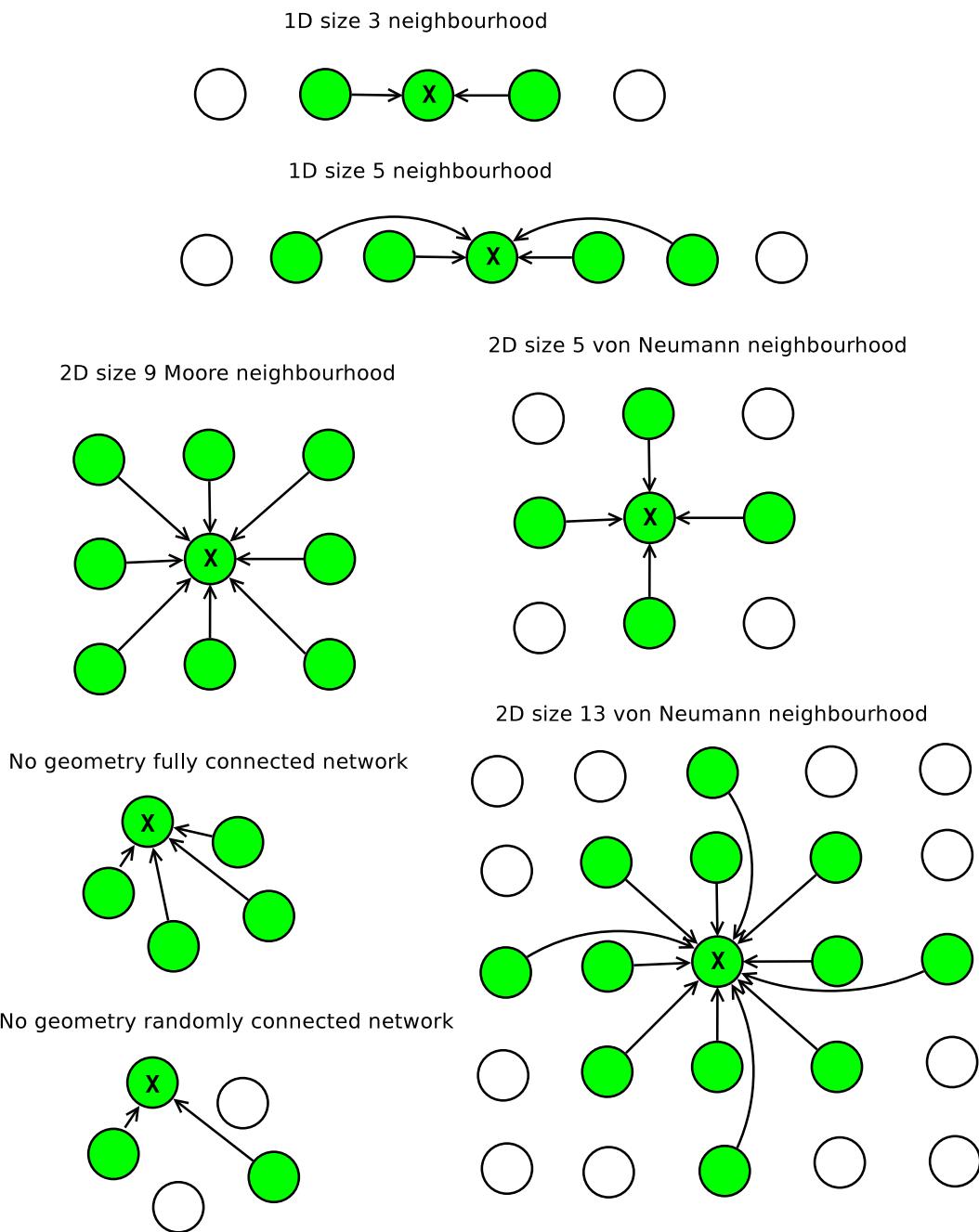


Figure 8.12: *The neighbourhood of a neural node is the set of nodes that directly influence its update, including the node itself. Here the node that will be updated is marked with an X, the neighbourhood nodes are green, and other non-neighbourhood nodes are white. Edges are plotted only from neighbouring nodes to the node that would be updated — in reality, this pattern of connectivity would be repeated for every node in the graph, but plotting this here would make the diagram unreadable.*

8.7 Neuron quantisation

In order to conduct experiments using neural network models it was necessary to implement those models in a programming language. The neuron models described in section 3.4 — the sigmoid, Beer, Taga, and Ekeberg continuous models, the spiking integrate-and-fire and spike response models, and a simple sine wave generator — were all implemented in the Python programming language. For each of these models, a quantised version of the model was devised, in which any floating-point (i.e. continuous) registers or arithmetic were replaced with equivalents taking or operating upon values with discrete levels. The quantised models were parameterised to allow the number of quantisation levels to be specified, rather than being fixed.

The implementation provided an abstract framework for constructing and simulating neural networks, where a neural network could, if necessary, arbitrarily combine different neuron types. At this stage, specific neural networks for creating specific robot morphologies were not created. Specific neural network instances would later be created by evolution using the genetic algorithm framework, which could manipulate neurons, neuron parameters, connections between neurons, and connection weights. In the case of a controller for a fixed robot, the neural network would be evolved as a single genome (see experiments in chapter 10). In the case of a co-evolution experiment, evolving both the controller and robot morphology together, the specific network would be evolved together with a morphology as part of a combined genome (see experiments in chapter 11).

An important point to note here is that any simulation carried out on a digital computer is inherently digital, and 64-bit floating point registers are actually digital (the values stored are rational), and not continuous. Hence, when we talk in this chapter and others about “continuous” models, this is a reference to the mathematical model, which is an idealisation. The implemented model that simulates this “continuous” model actually specifies a minimum of 64-bit floating-point registers (IEEE standard *double*), and in practice x86 processors were used, which possess 80-bit floating-point arithmetic units internally (the 80-bit data type is an *extended precision* defined in IEEE 754) [159]. In neural network research the use of standard x86 floating-point units to simulate so-called “continuous” models is commonplace.

8.8 Evolution of the neural networks

Neural networks are subject to evolution and optimisation by the genetic algorithm. Mutation was the only implemented genetic operator. The default mutation probability was set to 5% based on pilot runs; this means that, during reproduction of a genotype, the probability of a particular node, edge, or parameter being mutated was 5%. An elitist genetic algorithm was implemented, which preserves the top ranked (reproducing) individuals in each generation. For an elitist genetic algorithm it makes no sense to have children that are exact copies of their parents, since the parent genotype will already be present in the next generation. To avoid this situation the reproduction function enforced a minimum of one mutation in each child genotype.

Individual neurons had their parameter values mutated by replacement with either a completely random value, or with one drawn from a Gaussian distribution centered around the current value (for a description of the available neuron models and their parameters see section 3.4). The parameters subject to mutation by the genetic algorithm for each neuron type were:

Sine generator Period, phase offset, amplitude. The period determines the frequency of the sine wave. The amplitude determines the signal strength. The phase offset determines how the signal generated by this particular node is offset relative to 0 radians, in effect providing a global synchronisation scheme for all of the sine wave generating nodes in a robot's nervous system.

Beer's CTRNN Adaptation rate, bias. The adaptation rate determines how quickly the neuron state changes, the bias determines an *input* threshold for the neuron (this is the same as having an extra connection coming from a hypothetical neuron with constant 1 output).

Integrate-and-fire Adaptation rate, bias, firing threshold. Adaptation rate and bias are as in Beer's CTRNN model. The firing threshold is the value that the neuron's internal state must exceed before the neuron fires.

Spike response model Firing threshold. The other parameters (synaptic delay, membrane time constant and synapse time constant) were fixed to the recommended values for this model (see p.55).

Taga None. The Taga model has a published set of parameter values associated with it, and these were used as specified (see p.58).

Ekeberg Neuron type, and whether the neuron is inhibitory or excitatory. The neuron type is an integer which identifies a set of parameters associated with the Ekeberg model (see p.59). The actual value of the integer corresponds to the row in the published parameter table. The Ekeberg model is unique in having inhibition or excitation associated with a neuron rather than with individual connections, so this property is evolved.

Multi-value logical This neuron model relies on a lookup table to retrieve output values, so mutation consists of just randomly replacing entries in the lookup table.

The network structure is also subject to evolution and optimisation by the genetic algorithm, although to what degree depends on the type of network:

- The edges between nodes are weighted in every network type apart from those using logical nodes (because it does not make sense to “weight” a bit-string). The weights on these connections are mutated by random replacement with either a completely random value or one drawn from a Gaussian distribution centered on the current value.
- The connections themselves are fixed in all of the connectivity schemes apart from “random networks” (see p.233 for connectivity schemes). A random network will have a constant k value that specifies the number of incoming edges that each node has. In the initial population, k incoming edges are created for each node and randomly connected to source nodes. During mutation the genetic algorithm can add, remove and swap edge connections between neurons, but it must maintain the property that every neuron has k inputs. The parameter k is non-evolvable (it can be specified as part of the initial configuration of an evolutionary run, but is not subject to change by the genetic algorithm itself).

8.9 The software

The software developed for this project consists of thousands of lines of code. The main functionality is split between the genetic algorithm based evolutionary system, 3D physics simulator, an OpenGL based simulation renderer, and neural network simulator. Python was chosen as the primary development language as it has a good reputation for rapid prototyping, and wrappers for the underlying C/C++ libraries used were

already available. Despite its relative slowness as an interpreted language, early profiling showed that simulations spent the vast majority of their time inside the C physics simulator calls, and hence the alternative, to use a compiled language less amenable to rapid prototyping, was unlikely to pay off.

Genetic algorithms are known to be computationally intensive. This is due to the many thousands of fitness evaluations necessary for each evolutionary run; in a typical experiment a population of 100 individuals may be evaluated over 1000 generations of evolution, resulting in 100,000 fitness evaluations. To produce statistically valid results, it may be necessary to repeat this experiment 10 times, in turn producing a million fitness evaluations. Using a steady-state genetic algorithm may reduce the required number of evaluations by narrowing the search space, although almost all previous evolutionary systems for morphology and control have used a generational algorithm, with the exception of [298]. With a steady-state genetic algorithm there is no “next generation”; instead, a new individual is created from the existing population through crossover or mutation, this individual is tested, and if it is better than any of the existing members of the population one will be randomly chosen to be replaced. This has lower memory requirements, since only one working copy of a new individual needs to be stored, and produces a narrower search, since the new individual will only be kept in the population if it performs better than any of the existing population. With a generational generic algorithm each generation is replaced with the next regardless of the fitness of the individuals in the new generation — it is simply assumed that the fitness will improve over time — the exception to this being an elitist genetic algorithm, in which some defined top percentage of the population will be copied unaltered into the next generation.

The evaluation task is itself often computationally intensive. In this case, each fitness evaluation consists of a fully realistic 3D physics based simulation of many bodies interconnected with various joint constraints, and consideration of multiple collision points along their connected geometries.

Each simulation needs to be run over an extended period of time for accurate estimation of the controller or individual under test; 30 virtual seconds is typical for evolutionary robotics research. Using the above estimate of 1 million fitness evaluations produces a total requirement to simulate 30 million virtual seconds. If (as a very rough estimate) each virtual second corresponds on average to a single CPU time second (which is not unreasonable given several year old hardware), this single experiment would take 347.2 days of CPU time to complete. This is clearly unworkable

within realistic time constraints, and therefore a distributed system must be utilised to exploit the parallelism inherent within the evolutionary algorithm.

The systems architecture (what each PC connected to the network does) is shown in figure 8.13. The object database ZEO is used to store populations, genomes, and the fitness results from the evaluation function. The diagram shows the ZEO database running on a single server. Numerous client PCs from a computational cluster can connect to and access the ZEO server via the network. The user can start new evolutionary runs with specific parameters by using control software running on their own PC. The control software constructs a new population and pushes it and the associated genetic algorithm parameters into the ZEO database. The computational cluster clients are notified that there is new data, and will begin to run the distributed genetic algorithm, fetching genomes, evaluating them, and committing the results back to the database.

The software (figure 8.14) actually deployed on each PC is a mixture of C, C++ and Python, with appropriate wrappings for the C and C++ objects to make them accessible from the Python core. Note that this diagram shows the *software architecture* of the deployed software, whereas the previous diagram showed the systems architecture. The software architecture consists of a logical abstraction of the complete software suite which is distributed and installed on all of the computational cluster PCs and on the user's PC.

The front-end *ev* application has various command-line arguments that enable it to be used to carry out many tasks, such as the creation of populations, specification of genetic algorithm parameters, starting of the distributed genetic algorithm as a daemon process (for running on the computational cluster PCs), running the evaluation function on specific chromosomes, plotting graphs of evolutionary statistics, and running simulations and visualising the results in a 3D renderer. Individual simulations can be carried out using any genotype from the database, and these can be viewed using a custom OpenGL based renderer which allows the user to move about inside the 3D world, and change various rendering parameters interactively (such as plotting axes of rotation, endpoints, wire frame geometries, contact points etc.). The renderer can also record JPEG images, and produce MPEG-4 encoded movies depicting the simulation over time. The simulation engine allows various values to be recorded, such as the values of individual neural states, sensory inputs, and motor outputs, and these can be later plotted onto graphs.

It is important to ensure that the software functions correctly. It is well known that seemingly trivial bugs in software can cause significant errors in the output, and hence

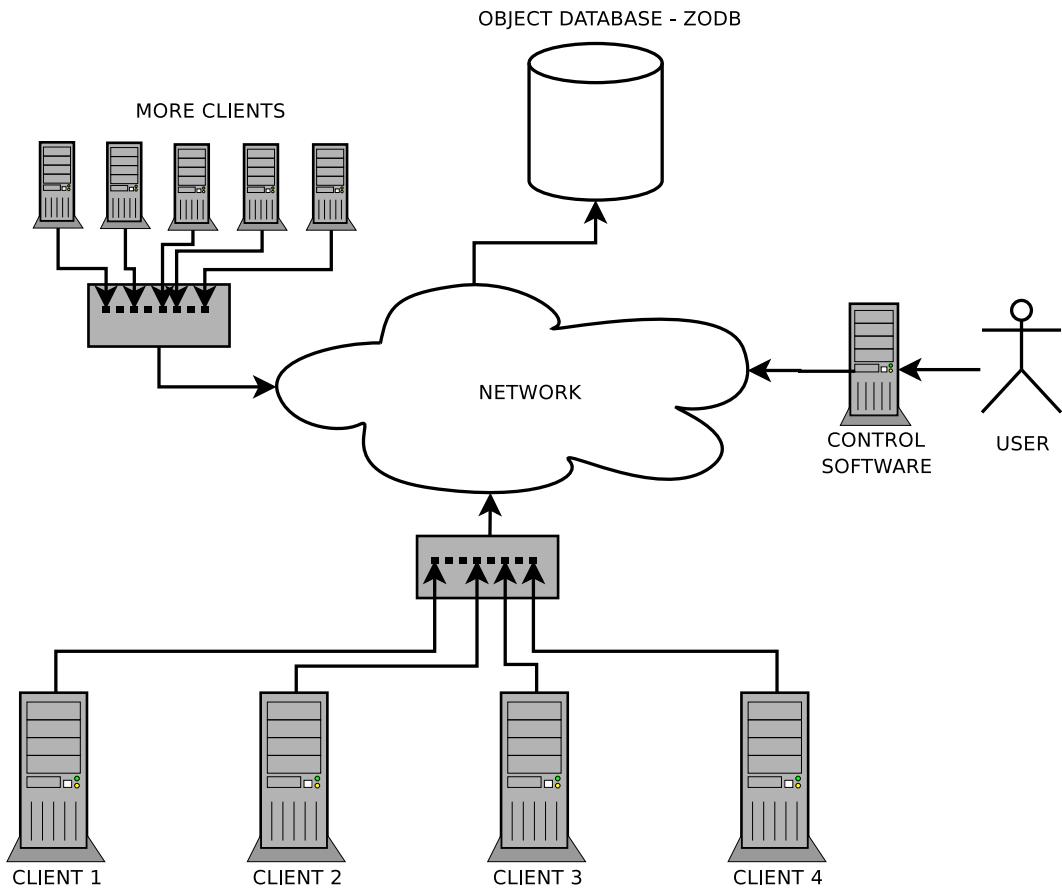


Figure 8.13: Network architecture of the evolutionary system. A distributed cluster of Linux based PCs is used to carry out the computationally intensive physics simulations. The centralised object database stores genotype populations and fitness values from evolutionary runs. A custom control application allows the user to configure and start experiments.

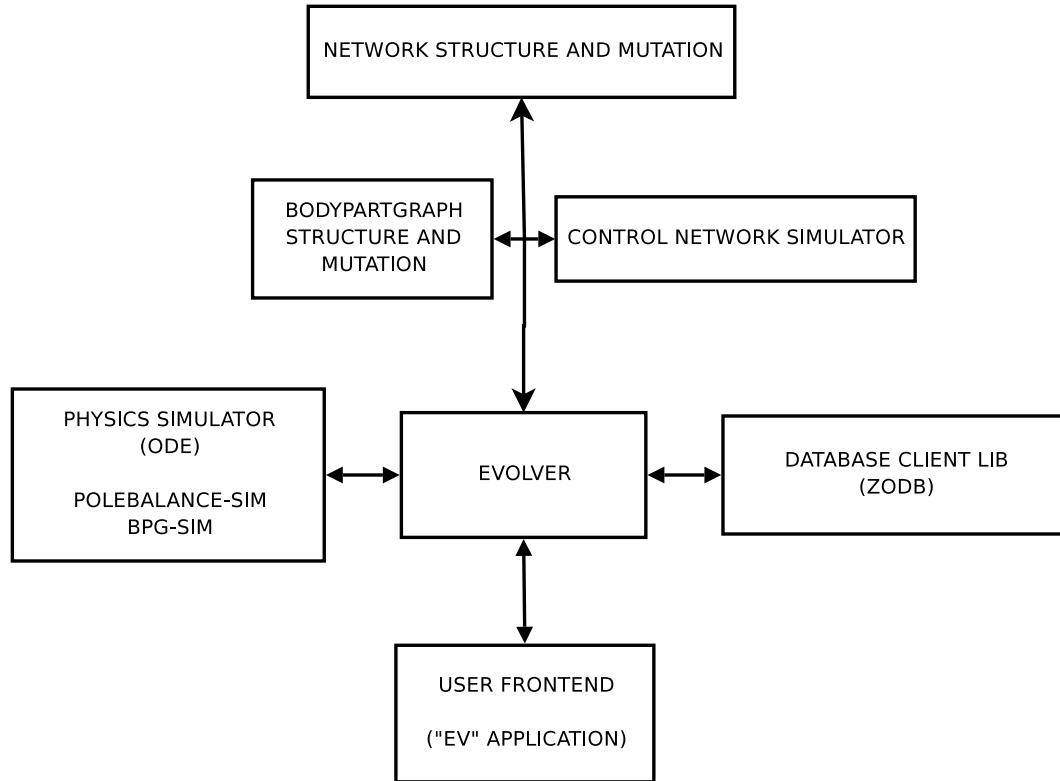


Figure 8.14: *The software architecture of the evolutionary system. This shows the logical abstraction of the software suite into different modules with different functionalities. The whole software suite is distributed and installed on every PC in the computational cluster, and on any PCs that the user will use to control the evolutionary process.*

software must be well tested. In projects like this, that rely completely on the results of simulation, there is no guarantee that the results are transferable to the real world, although the success of similar evolutionary robotics projects shows that transference is certainly possible. For more on the limits of testing see section 1.3 for a discussion of reproducibility and other general testing problems, and section 9 for information on the testing of neuron models.

In order to test the software over 200 units tests were developed; these test everything from the instantiation and operation of a single neuron, to complete evolutionary runs consisting of a number of individuals over several generations. Unit testing is a standard software engineering methodology, and it is well recognised that units tests, whilst useful, can only exercise a small number of paths through the software under test — they do not (and can not) guarantee that a function is correct for all inputs. They are devised by manual inspection of the source code, and attempting to devise functions that will exercise blocks of code and functionality independently. These tests were used for regression testing of the code under test — in practice, the code and the test must function together, which implicitly exercises the tests. The test suite was run after the source code was modified and before being checked in to a revision control system in order to ensure that no bug regressions had been introduced by the recent changes. It is widely recognised in the software industry that integrating a unit testing regime with the development process generally leads to improved code reliability.

Although simulations and evolutionary runs have an element of randomness, this randomness derives completely from an initial random seed, so that any particular experimental result should be completely reproducible. With the current code, simulations are 100% reproducible — the same seed and individual will always result in the same fitness score, but due to the use of non-deterministic Python operators in the evolutionary functions (such as iterating over dictionaries), complete evolutionary runs will not be reproduced exactly. This small issue is fixable with some effort if desired in future, and is highly unlikely to significantly affect the experimental results presented here, since the results represent the aggregate data of many individual runs — the non-deterministic operators will affect individual runs, but will just be another randomised parameter that is essentially meaningless in the context of the entirety of the experimental results.

8.10 Simulated physics

There are two components to a physically accurate simulation — the dynamics simulator and the collision detector. The “Open Dynamics Engine” (ODE) [404] physics library was used to run simulations and perform collision detection. ODE simulates rigid 3D physics with physically realistic movement, contact, and friction models. It was developed with fast simulation for real time video games in mind, which means that there are a few known problems with accuracy (the manual warns that simulations are not “industrial quality”). Despite this, it is widely used in evolutionary robotics research, for evolving morphologies [118, 359], evolving controllers for fixed morphologies [9, 41, 44, 104, 163, 173, 281, 282, 363, 430, 431, 457, 493, 494], evolving parametric morphologies with control [180, 461–463, 496], evolving modular robots [42, 276, 468, 469], evolving multi-agent emergent behaviour [280, 450], evolving control and morphology [40, 60, 61, 295, 297, 390–393], and also for non-evolutionary robotics [419, 420, 488]. It is also used as the physics back-end behind the robot simulation package “Webots” [294], and the artificial life simulation environment “Breve” [234].

Visual rendering was performed using a custom written OpenGL application. Although a generic geometry rendering engine could have been used, the custom application is specialised towards the task of displaying evolved creatures, and can plot contacts and the axes of individual body parts, which was a great help while fixing bugs and analysing simulation accuracy.

The dynamics simulator allows the user to create particles with mass in 3D space, and apply forces to them. It uses a first order integrator to generate physically realistic motions of the particles. Particles can be connected via joints to create articulated bodies. The joints, which themselves occupy a point in 3D space, impose constraints on the degrees of freedom of particles they connect. The joints available include ball and socket, hinge, and universal, providing a range of possible behaviour.

The dynamics simulator is called with a step size parameter, which specifies the time that will elapse during this step. A large step size produces unrealistic behaviour, as the vector gradient for each body is constant over this time, and also inter-penetrations will occur as it is not possible to perform collision detection until after the dynamics step. Too small a step size is computationally intensive and will also produce unrealistic behaviour since the changes in particle positions will be small or non-existent, which prevents the integrator from accurately calculating motion gradients. For con-

venience, the step size used is usually the same as the rendering frequency, which for high-end flicker-free animation is usually 50Hz or 60Hz; these exact numbers are a legacy of the PAL and NTSC TV standards, which were in turn the result of research on motion perception of the human visual system which found that a sequence of still images displayed at these frequencies will be interpreted as continuous movement, so the resulting animation appears to be realistic. An update frequency of 50Hz (20ms step) has been commonly used for research carried out with the ODE physics engine, and so it was decided to use the same frequency in this research.

The collision detector allows the programmer to associate geometric primitives, such as cuboids and spheres, with particles. After the dynamics step the new positions of the particles are known. The geometry of these particles can then be checked for intersections, which indicate that a collision has taken place. To prevent the bodies from passing through each other a contact constraint needs to be created for each collision, applying an instant restitution velocity to both particles, so that after the next dynamics step their geometric primitives will no longer intersect.

Geometric bodies are fashioned from different materials with different properties (such as coefficients of friction and restitution). When a contact occurs the material properties of the two colliding bodies are used to calculate appropriate friction and restitution forces along some two-dimensional vector on the contact surface.

8.10.1 Collision detection

The robot was subject to standard gravity, so collision detection between the robot and the ground plane was necessary to prevent the robot falling through the ground. Collision detection was not performed between the cylinders that made up a creature's body. Since the capped ends of connected cylinders are centred on the same point and the hemisphere caps rotate through the same spherical space there is no point in performing collision detection between the actual caps of two connected cylinders because they would always be in contact. A system where the connected cylinders could collide but the end caps could not was implemented, but pilot runs failed to evolve any locomoting creatures — presumably not allowing body parts to interpenetrate too heavily constrains the morphology for such a simple evolutionary system, although it is possible that with more work a morphology schema that disallows inter-penetrations would be viable. This is not the only work evolving control and morphology that has faced this issue — Sims's work allowed connected parts to interpenetrate but not com-

pletely rotate through each other [398]. Ray noted that he had experimented with two different forms of collision detection, but that neither had produced satisfactory results, and so he ran his experiments with the collision detection turned off [348]. Vaughan also reported that in his evolved biped walker there was no collision detection between the two legs [462].

8.11 Sensors and actuators

The simulated robot has motors on joints which enable it to move. These motors are connected to outputs from the neural controller. The robot is capable of perceiving a small amount of information about its environment. It has proprioception (information about the body's internal state) from sensing the angles of its limbs, and exteroception (information about the world) from its spherical feet which can sense whether or not they are in contact with the ground.

8.12 Motor models

Three different models were used for joint motors, two based on the *Open Dynamics Engine* (ODE) internal motor model, and one based on directly applying a torque. The ODE motor internally works by adding additional constraints to the physics simulation, and allowing its “linear complementarity problem” (LCP) solver to work out a solution minimising global constraint error (joints are also modelled as constraints).

For all motor models, the output of a neuron had to be converted into some value to be input to the motor model. For continuous neuron models this simply involves normalising the value. For spiking neuron models the issue is a bit more complex, as the discrete spikes must be converted into a real value. Section 3.3 described several explanations that have been put forward to explain how signals are coded in biological networks, and how signals have been coded in previous research into synthetic neural networks. The approach taken in this work was to normalise the internal state of the output neuron. This avoids the question of how to decode the outgoing spike train. The internal state is almost a continuous representation of the spikes anyway; if we assume that the function of a spike train decoding process is to regenerate the internal state of the transmitting neuron, then this would in effect be the perfect decoding function. Such an assumption is not terribly unrealistic, and given that we have no explanation

for how biological spikes trains are actually coded, this seems a reasonable approach to take for the research in this thesis.

The first controller interpreted neural network output signals as the desired velocity of the motor in the range 0 (reverse), 0.5 (off), and 1 (forward), i.e. the desired velocity was scaled from [0, 1] to $[-VMax, +Vmax]$. The actual rotational velocity of the joint was constrained by the attached masses. A maximum force F_{max} was specified, which limited how quickly the motor could react to changes in the desired velocity. During pilot runs it proved quite hard to evolve creatures using this model, presumably because an oscillating velocity does not directly translate into oscillating movements.

The second motor model was a proportional derivative (PD) controller. Again the signal was in the range [0, 1], which is the output range of the simulated neurons. This signal is scaled to the range $[-\frac{\pi}{4}, \frac{\pi}{4}]$ and used as the desired limb angle. The actual desired velocity input to the internal motor is calculated as:

$$T = K_p(\theta_d - \theta_a) + K_d \frac{d(\theta_d - \theta_a)}{dt}$$

where K_p is the proportional constant, θ_d is the desired angle, θ_a is the current angle, and K_d is the derivative constant. The proportional and derivative constants were experimented with by trial and error until values that generated realistic motion were found.

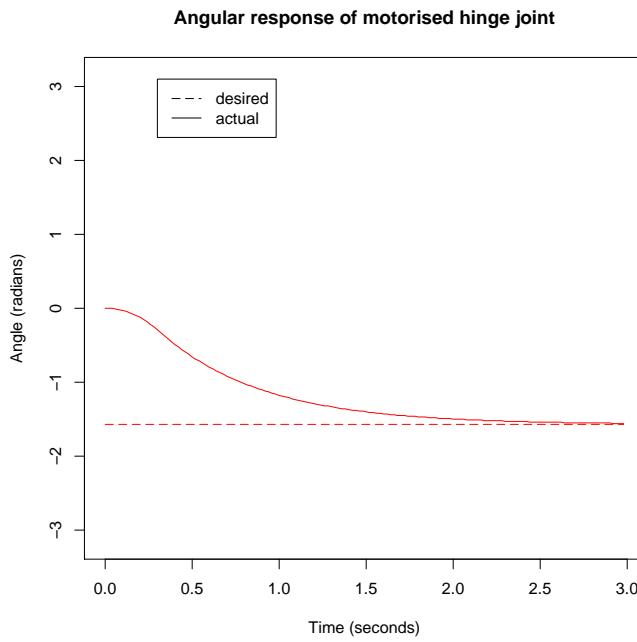
The third controller was identical to the second, except that instead of specifying a desired angle for the internal motor, an instantaneous torque was directly applied to the joint at each time step. The proportional and derivative coefficients were again chosen by trial and error. This controller worked successfully when unit tested against pairs of bodies attached by hinge, universal, and ball joints, but when used in actual evolutionary runs simulations quickly became unstable. The ODE manual warns that this might happen when directly applying forces to joints, and recommends using the internal motor model specifically for this reason.

8.12.1 Stimulus-response curves of PD motor controller

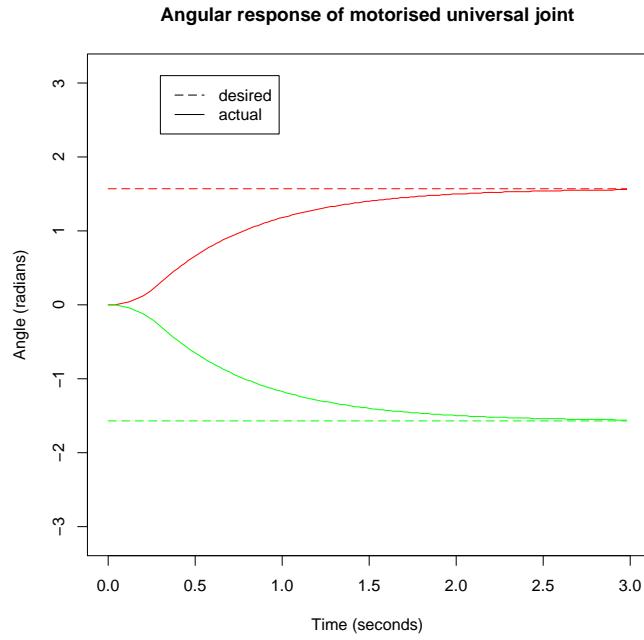
Motor controller stimulus-response curves show how the motor controller changes the current angle of a joint to move it towards some desired angle, or in the terminology of control theory, how the *error* is reduced (where error is defined as the difference between the desired angle and the current angle). The joint can have either 1, 2, or 3 degrees of freedom, corresponding to the “hinge”, “universal” and “ball” joints of the

Open Dynamics Engine simulator. The aim of these plots is to examine how quickly the joint moves, whether or not the joint approaches the desired angle, and whether or not the joint stabilises at the desired angle (as opposed to overshooting it, or oscillating around it).

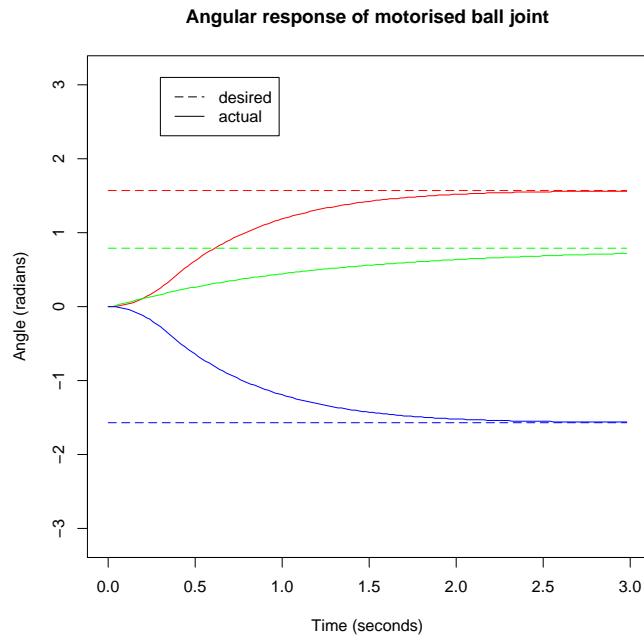
We are plotting the velocity based proportional derivative controller because pilot runs showed it to be the most successful motor model in evolving creatures. The shape of the response curve indicates the process the controller uses to approach the desired joint angle — in this case, the curves should indicate aspects of proportional derivative control, i.e. velocity will be higher when further away from the desired angle, and will decrease rapidly upon approach, producing an initially quick correction of the error that slows over time. Each simulation started from a static position. Standard gravity was not simulated for this test.



The error (difference between desired angle and actual angle) of the hinge joint is quickly corrected. The curve shows that the initial velocity is higher than later velocity. No overshoot occurs.



The error is quickly corrected in both axes of the universal joint. The curves are almost identical to that of the hinge joint.



The motor must correct the error along all three axes simultaneously when driving a ball joint. The shape of the curves indicate that the motor controller is working as expected. The response curve indicates that it is somewhat slower to correct the error along the second axis of the ball joint (plotted as a green line). Attempting to increase

the speed of the motor along this axis led to instability problems with the motor model, which resulted in failure to correctly converge on the desired angles of all three axes, so the model was left as is.

8.13 Physics simulation problems

There can be problems with both the accuracy and stability when modelling multi-body dynamic agents with a physics simulator. The “open dynamics engine” converts the user specified model into a series of constraints, and uses a “linear complementarity problem” (LCP) solver to calculate updated positions for bodies with mass whilst minimising constraint error.

The simulation of evolved agents can cause particular problems; unlike manually designed robots, joints with conflicting constraints are more likely to occur, and designs are more likely to be non-symmetric, and have unusual combinations of materials with different mass, shape, size, and friction coefficients. All of this non-homogeneity increases the probability of numerical error and instability.

Two values, in particular, trade off accuracy and stability. The “error reduction parameter” (ERP) specifies the proportion of constraint error that is corrected in each time step. The default is 0.1, meaning that one tenth of the error of each constraint will be corrected. The “constraint force mixing” (CFM) parameter specifies the degree to which a constraint can be violated (or the “hardness” of a constraint). The default is 0.1, which means the constraint can be violated by one tenth of the force necessary to perfectly correct it. Allowing constraints to be soft like this increases the probability of being able to find a state that simultaneously satisfies all of the given constraints and reduces the chance of the simulation “blowing up” due to a singularity.

The physical simulation of evolved dynamic morphologies is subject to numerical instability which can cause the simulation to behave in an unrealistic way. One symptom of this is the “explosion” effect originally noted by Sims [397, 398], in which numerical instability causes the simulation to “blow up”; this can be caused by a singularity, opposing constraints, very high velocities, and constraints between bodies with large differences in mass. Sometimes the solver adds force to bodies, but this results in a greater violation of the constraints, so the solver attempts to use more force, and a feedback loop is created which will eventually force the simulated bodies to explode away from their composite centre of mass.

Another effect noted by Sims was that evolution tended to exploit inaccuracies in

the simulation in order to create higher fitness populations. For example, unrestrained rotation of two bodies about their connecting joint, in the absence of joint friction, will cause the bodies to gain rotational velocity relative to each other. There are no upper limits on this velocity, and due to simulation inaccuracy this rotation can cause the composite centre of mass of the inter-connected bodies to move in 3D space. Hence rotation without friction can be utilised to directly transfer energy from motor activity to body velocity. This can be exploited by evolution; Sims observed the development of agents which would jump, spin in the air where they are subject to no friction, gather velocity, and be projected flying through the air.

Different solutions have been used to control the “explosion” effect. Many fitness evaluation frameworks are programmed to report that simulations in which the bodies achieve velocities above some threshold are invalid, such as [60, 61, 457, 468]. This was the approach originally taken in this research. However, it proved ineffective at controlling the exploitation of the instability. When a threshold value was used the evolutionary algorithm would tend towards creatures that lie just below the threshold, allowing it to exploit the instabilities to generate physical movements, and yet not display the physical explosion effect. When the threshold was slightly higher, this tendency could be observed visually; agents would explode, but not attain significantly high velocities, and then the bodies would almost instantly re-converge. This behaviour would repeat in a never ending cycle.

This behaviour was also observed by Ray, who described it as it as generating motion “greater than any creature in the series up to this point”. He selected an exploding creature for reproduction, and then evolved it through successive generations to temper the explosion. It was “perfectly balanced on the edge between exploding and pulling back together. Soon after birth, the flat panels begin shaking... the whole collection of parts scatter and begin spinning as if in a whirlwind. When the scattered parts have become completely disordered, the flat panels may begin shaking again, and the shaking somehow pulls the parts back into their original positions” [348].

In this research it proved impossible to find a threshold that worked in all cases. Even when the system was restricted only to velocities that could realistically be generated by the motors, evolution would still exploit violated constraints between the many bodies, joints, and geometric primitives that make up a creature. This was proven by using a simulation model in which the motors were inactive. The creatures, previously evolved with active motors, still displayed the same locomotion behaviour, thus showing that the motors were not responsible for the creature’s movements. This left only

the violated constraints as a source of energy in the system.

In order to minimise this effect a “relax” phase was added to the beginning of each simulation. For some number of seconds the motors are inactive. During this time the agent will fall due to gravity and impact the ground. Since it has no motor activity, it should fall and come to a complete rest on the flat surface. If it does not come to rest, then movement is being generated by violated constraints, and the simulator returns an invalid genotype value to the evolutionary algorithm, which in turn heavily penalises the genotype responsible. With this system, it is still possible for a valid genotype to result in constraint violations, but the evolutionary search is now biased and will avoid these areas of the solution space.

It is interesting that other researchers have not noted and corrected for this effect. Either their morphologies are tightly constrained and their physics simulators do not have violated constraints, which seems unlikely when evolving arbitrary morphologies, or the effect has been present but has gone unnoticed, which may be more likely — unless the motor models are completely deactivated this effect is relatively indistinguishable from movement generated by the neural control system at velocities below the explosion threshold.

One of the factors contributing to the explosion problem was that air resistance to moving geometries, and dry friction in joints, were not simulated. Force could be added to rotating bodies until their velocities became too high and the robot would spin itself apart. This was due to the lack of restraints on the angles of joints — they could freely rotate about 360° . To try and counteract this evolvable stop limits were added to each joint axis, preventing the joint angle from exceeding an upper and lower limit. It was still possible for the evolved limit to be infinite, which effectively removed it, so the problem was not totally fixed.

In the end the evolvable limits were removed, and angular stop limits became mandatory for all joints. This revealed another problem; the stop limits are treated as an ordinary constraint by the solver, so they can be violated in order to reduce the global error. Once a stop limit had been sufficiently violated the body would pass the limiting point, and that limit would no longer apply. Again, as there was an evolutionary pressure towards movement, this would result in the evolution of individuals with joints rotating through 360° .

According to the ODE manual (and comments in the source code), motor force acting against a stop limit within the same joint should be detected and nullified. However, this does not appear to be the case when an angular motor is attached to a joint,

even when the limit is applied to the motor and not joint. To fix this the software had to be changed to ensure that the desired velocity given to the motor model falls to zero as the limit is approached, thus preventing the motor model from applying force against the limit. Despite this, the system could still evolve creatures which violated the stop constraints. Shifting the stop limits to $\pm\pi$ radians, so that they were at the same point, rather than symmetrical about this point, seemed to bring more stability (symmetrical stops about π radians seemed to introduce some unstable singularity). A minimum number of body parts was enforced, as too few body parts contributed to energy gain and violation of stop limits. Despite all these measures, a few evolved creatures still manage to violate the stop limits and rotate their joints through 360° .

Initially, the axes for a ball joint were evolved. It was hoped that this would allow the evolutionary algorithm more freedom in exploring the solution space, and enable the development of unique joints in which the axes of rotation did not correspond with the angular rotation of either of the connected geometries, allowing manipulation of a single axis to produce arbitrary rotations. Simulations with this property were often unstable, and investigation led to the discovery that rotational momentum about the second axis of a motorised ball and socket joint does not seem to be conserved, whereas it is around the other axes. To solve this the axes were fixed with respect to the parent body, and the controller for the second axis of rotation was given a different (much smaller) proportional coefficient.

Chaumont noted that the motor models of ODE are vulnerable to undesired oscillations [60]. He corrected for this by connecting the outputs of the neural network to F_{max} of the motor model, F_{max} being the upper limit of force that will be applied by the motor to the joint in any given time-step to control the velocity. This effect was not noted in this research, however the motor models would have had different parameters, and possibly been in a different operating mode.

The CPU time required to run a regular ODE simulation has order $O(N^3)$, where N is the number of constraints in the simulation. This can severely slow running time when a large number of connected bodies are present in the simulation (i.e. when a robot has many body parts). This was a problem with unconstrained evolution where there was no penalty (and perhaps some kind of implicit reward) for increasing the number of body parts — figure 8.15 shows an example unconstrained morphology that causes the OpenGL driver for a particular brand of graphics cards to crash. To counter this, a constant upper bound on the number of body parts was set; a practical value of 20 was used on a pentium-M laptop, but the value will vary depending on the use of the

simulation (e.g. slow simulations may be more acceptable when real-time interaction is not required) and computational power available.

Alternatively, ODE provides a “quickStep” algorithm which is of order $O(N)$. Simulations using this function trade-off physical accuracy for simulation speed. Some pilot runs were carried out where evaluations were alternately carried out using the two functions, with the aim of increasing the speed of simulations. However, the fitness functions used in this research, which aim to maximise robot velocity to encourage the development of locomotion, generally implicitly penalise the development of useless extraneous body parts as they increase the overall mass, but do not contribute to increased velocity. Hence, the number of body parts tends to remain low, and no great gains were seen in simulation running time with the “quickStep” algorithm.

A similar problem was discovered with the growth of memory usage; again, unrestricted evolution can produce a variety of morphologies which when simulated have an unknown number of constraints. The regular “step” algorithm has memory requirements which grow with order $O(N^2)$. The “quickStep” algorithm has order $O(N)$. The ODE library uses its own memory handling routines which are optimised for speed — they allocate from a pool of memory located in the executable stack rather than call the generic C/C++ operators which would dynamically allocate memory on the heap.

This means that simulations are limited by the stack size available to processes on the operating system; it may be possible to increase this up to some maximum, but this depends on the policies and current state of the OS. Unfortunately, when the stack size is exceeded ODE will crash the running process, rather than detect the problem and return an error to the calling function. This is an extremely irritating behaviour, as it initially appears to indicate some programming error, but is actually just poor handling of an out-of-memory condition. Figure 8.15 shows an example unconstrained morphology that can not be simulated using a standard size stack, as the large number of ODE primitives causes a stack overflow.

A patch to use the regular memory allocation routines is available, but it applies to an older version of ODE and does not seem to be widely used. Limiting the maximum number of body parts and increasing the maximum stack size to 64MB (the largest possible on x86 Linux) seemed to prevent the problem in this case. In Linux the stack size of a process is not fixed, it can dynamically grow towards this maximum, but can not exceed it. This means that it is still not possible to compute simulations with the standard ODE library that require more than 64MB, although simulations larger than this may be prohibitively slow anyway. When using a cluster with other users some

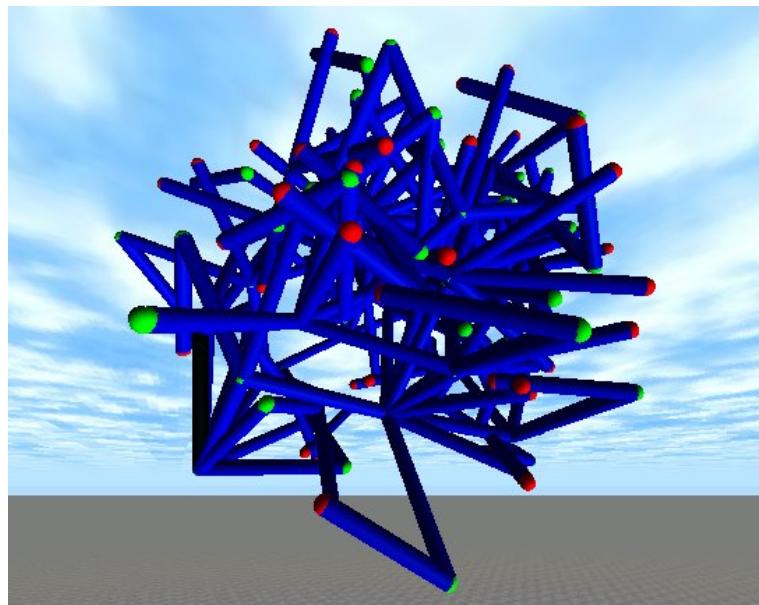


Figure 8.15: *Unconstrained evolution creates morphologies that stress computational abilities. This design with 700 body parts can not be physically simulated with the regular ODE library as it overflows the 64MB stack (the maximum on x86 Linux). Visualising this design was also a problem; the irregular geometry and high number of primitives produces a segmentation fault from the proprietary ATI video card driver (this screenshot was captured under a debugger).*

of the client hosts may not allow the user to increase the maximum stack size, and these processes will subsequently crash when faced with a large simulation. This is inconvenient as these processes must be restarted, but in general, as long as at least one process has a large maximum stack size, the genetic algorithm will eventually be able to proceed.

Some researchers have hypothesised that one way to make evolved robot designs more robust is to evaluate their fitness functions using two (or more) physics simulators, and then use either the mean or minimum fitness value. This strategy aims to prevent the evolved robots from exploiting inaccuracies, or bugs, in any particular physics simulator, and hence make the evolved robot designs more robust to variations in the physics environment, and more likely to transfer to reality successfully. Some pilot runs were carried out utilising this strategy, by using the “quickStep” algorithms for two-thirds of the evaluations, and the regular “step” algorithm for the other third. However, this seemed to actually inhibit the evolution of successful designs; this is probably due to the variance in simulation accuracy between the two algorithms being

too large for this particular type of simulation. The “quickStep” algorithm has poor accuracy for near-singular systems, which includes those using motors and having many bodies with “looping” style connectivities, which is precisely the kind of multiple-legged robots, with each foot being simultaneously in contact with the ground, that we are likely to evolve.

8.14 Tasks

A number of fitness functions define the type of tasks the robots can be evolved to carry out. These include measuring the mean height of the robot over time, the mean distance over time, the sum of motion between frames (to encourage movement), and a combination of the movement and the mean distance functions (to encourage walking).

Reeve compared fitness functions for evolving walking behaviour and concluded that a simple distance measurement was one of the best. Combining simple penalties with a decaying weight function that values earlier motion higher than later motion resulted in a slight improvement [349]. This suggests that the mean distance over time function may be as successful as the more complex movement and walking functions.

8.15 Summary

Custom software has been developed to carry out the experiments described in this thesis. The software primarily relies on the *Open Dynamics Engine* physics simulator which has been widely used in previous robotics research. An OpenGL application enables evolved creatures to be visualised in a 3D simulation, with features such as rotation, zooming, pausing, logging of neural activity for later analysis, and capture of JPEG still images and MPEG video. The evolution software itself was custom written for this thesis, and makes use of a distributed implementation enabling hundreds of PCs to be used simultaneously. Various neural network models have been implemented, along with quantised versions of those models that use discrete states and integer arithmetic. This will allow a comparison of these neuron types and quantisation models to be made. The software allows various other features of the experimental setup to be varied, such as the neural network configuration, neural timing model (synchronous or asynchronous), population size, and number of generations to evolve, enabling experiments to be carried out that will determine whether these factors contribute to the overall performance of the evolved control networks.

Chapter 9

Software testing

Software testing is performed to ensure that the software is stable (does not crash), and to attempt to ensure that it functions correctly. It is difficult to comprehensively test any piece of software (for a wider discussion, see section 1.3). In order to test that the software itself could be used as part of a larger program without crashing, or causing memory leaks or other problems, a large unit test suite was created (see page 240 for a description of unit testing). Unit testing is a standard software engineering process that is recognised as generally improving code quality, but of course, it can not test for all possible errors. Unit tests exercise particular code paths and data sets, they do not, and can not, test all of the possible combinations of ways in which the software may be used, and so can not guarantee that the software will function correctly under all given use cases. Despite this, unit testing is useful in finding bugs and bug regressions, and in ensuring that the basic functionality of the code does not exhibit any obvious errors under the tested use cases.

One common method of testing is to present a predetermined input sequence to the device (or code) under test and observe its response. If the actual response matches the expected response the test is passed. This method is problematic when the element under test has some internal state holding logic, where the current output depends on an internal state which has in turn been determined by previous inputs. In digital logic, functional testing is sometimes done by comparing the output sequence to a published sequence. This is only possible when the algorithm has several implementations which can be used to verify the published sequence is correct, or when the published sequence has been manually verified by independent analysis. This kind of testing is not possible here, as there are no published input/output data sets for these neuron models. The best that can be done is to generate some input data, use the model to generate output

data, and then manually verify by inspection that certain events cause the response that we would expect. For example, we would generally expect that a large amount of input activity would lead to a neuron eventually exceeding its internal threshold and firing. After firing, we would expect to see the neuron entering its refractory period, preventing the internal activity from increasing for some time. Another simple test might be to strongly suppress neuron activity via an inhibitory input, and to verify that the internal state is lowered and no output spikes are generated during the time of suppression.

Another possible set of tests enables comparison of neuron behaviour between models utilising arithmetic with floating-point precision, or quantised arithmetic with various levels of discretisation. To test this, a particular neuron model using a particular type of arithmetic will be instantiated, and the instance will then be presented with some predetermined input signals that vary over time. The neuron's internal state and output signal will be recorded as the input signals vary. The recorded output data can then be plotted and manually observed to check that the signals are more or less consistent across the different quantisation levels of a particular model. What we would expect to see with a regular neuron model is the floating-point implementation providing the highest level of resolution and detailed activity, and the level of detail decreasing as the number of quantisation levels (and thus the arithmetic precision) decrease. It is possible that decreasing the arithmetic precision could change the behaviour of the neuron, which is undesirable — this thesis aims to show that quantised models can approximate the behaviour of floating-point models for robot control tasks — so if radical changes in behaviour are observed between the floating-point and quantised models of a single neuron, then similar radical changes in behaviour are likely to be observed when simulating full networks.

These tests are obviously quite limited, they do not and can not test all responses of the various implementations to all possible input signals. What they can do is verify that the software does not contain any obvious programming errors, verify that the implementations respond in some expected way to a change in input signals, and verify that there is some similarity in the behaviour of the implementations with different quantisation levels, and with the floating point implementation. Ensuring functional correctness of software is a notoriously difficult problem. All research that relies on software faces the problem that a single bug in the software could render the research useless. There is no way around this — the data presented in this thesis does rely on software — as does the data presented in the vast majority of evolutionary robotics

research by past experimenters. A discovery of a bug, such as a hypothetical bug in the *Open Dynamics Engine* physics simulator, could render invalid a large amount of the research that has relied on the integrity of such software.

9.1 Testing neuron models

To test the models a varying stimulus was applied to each, and the response recorded. The quantised models were tested with 1 to 6 bits of precision (i.e. 2, 4, 8, 16, 32, and 64 quanta states). The continuous version was also subjected to the same stimulus and its response plotted. For the first 0.5 seconds the stimulus signal was zero. From 0.5 seconds to 1.5 seconds a positive stimulus signal was applied via an inhibiting connection (or, in the case of the Ekeberg model, an inhibiting neuron). After 1.5 seconds the connection was switched to excitation, but the stimulus was still positive. Therefore, we would expect to see activity of the neuron under test declining for 1.5 seconds, and then subsequently increasing. Between 4 and 5 seconds there was no stimulus, so the neuron under test would either be devoid of activity in the case of the sigmoid model, or, in the case of the leaky models with internal state, would tend towards zero.

The following graphs show this stimulus/response behaviour; it is apparent that as the precision of the discrete levels was increased, the response waveform more closely approximated that of the continuous model.

The test regime here is one of manual verification that the waveforms are “somewhat similar”. This is a loose definition, as it is not required that the waveforms be identical, or even that they actually are similar. As the quantisation level is reduced, it is entirely possible that the behaviour of the neuron could change, generating a substantially different output. This would not indicate a failure of the test, but would indicate that larger networks of neurons at or beyond this level of quantisation may not behave in the same way as a similar network of floating-point neurons would. The fact that a neuron may display similar behaviour across a range of quantisation levels does not prove that the behaviour is actually the same, or that similar behaviour would occur if the input stimulus waveforms were different.

As previously recognised, the test regime here is limited. It is possible that lowering the arithmetic precision from floating-point to some limited integer range will produce a set of neuron models with completely altered behaviour. The question that may be answered here is: “Is it possible that reducing the arithmetic precision will not

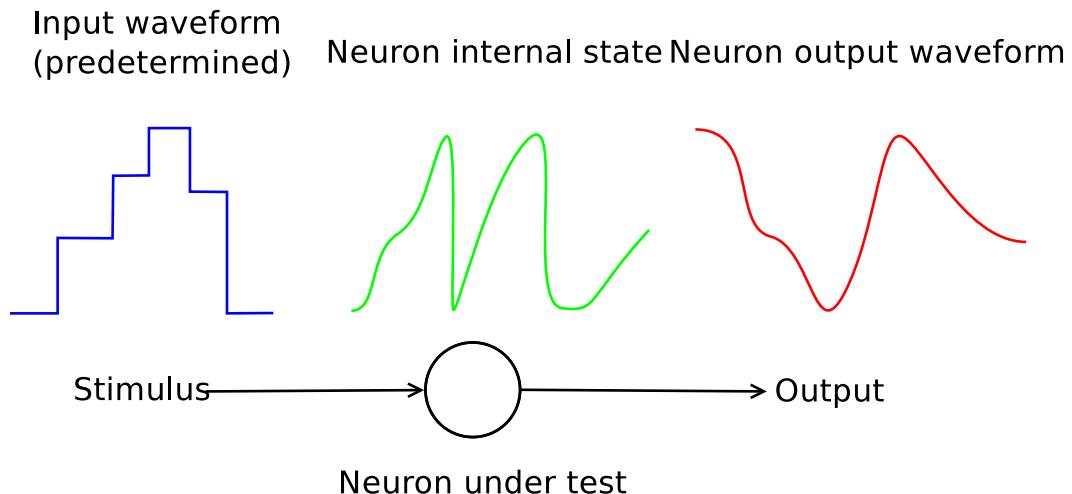


Figure 9.1: *Testing is done on a single neuron. A predetermined waveform is input to the neuron and the output waveform from the neuron is recorded.*

result in radical changes to behaviour?” A sequence of plots that illustrate that similar behaviour can occur with lowered precision would indeed affirm that it may be possible to lower the arithmetic precision without completely destroying the validity of the model.

Note that the plots here show input values to the neurons that are not quantised. In the software code used here, the quantisation occurs at the point where the neuron receives the value. For example, the “sine 2 state (1-bit)” plot uses two quanta states but the complete five levels of stimulus are visible; this is an artifact of using the same stimulus waveform for each test, and does not affect the simulation — the interpretation of this input value by the neuron will indeed be binary.

9.1.1 Explanation of graphs

Figure 9.1 shows the test setup and rendering: the input waveform (blue) is presented to the neuron under test, which might have some internal state (green), and which produces a single output (red).

Blue line Stimulus is the blue line. This is a predetermined waveform connected directly to a neuron input.

Green line The internal state of a neuron is represented (where practical) by a green line.

Red line Neuron output is the red line. This depends on the neuron's response to the input waveform. The red line will be continuous or spiking depending on the neuron model.

off | inh | exc | off State of stimulus. The stimulus transitions through states (off, inhibitory, excitatory, off) at times (0.5s, 1.5s, 4s).

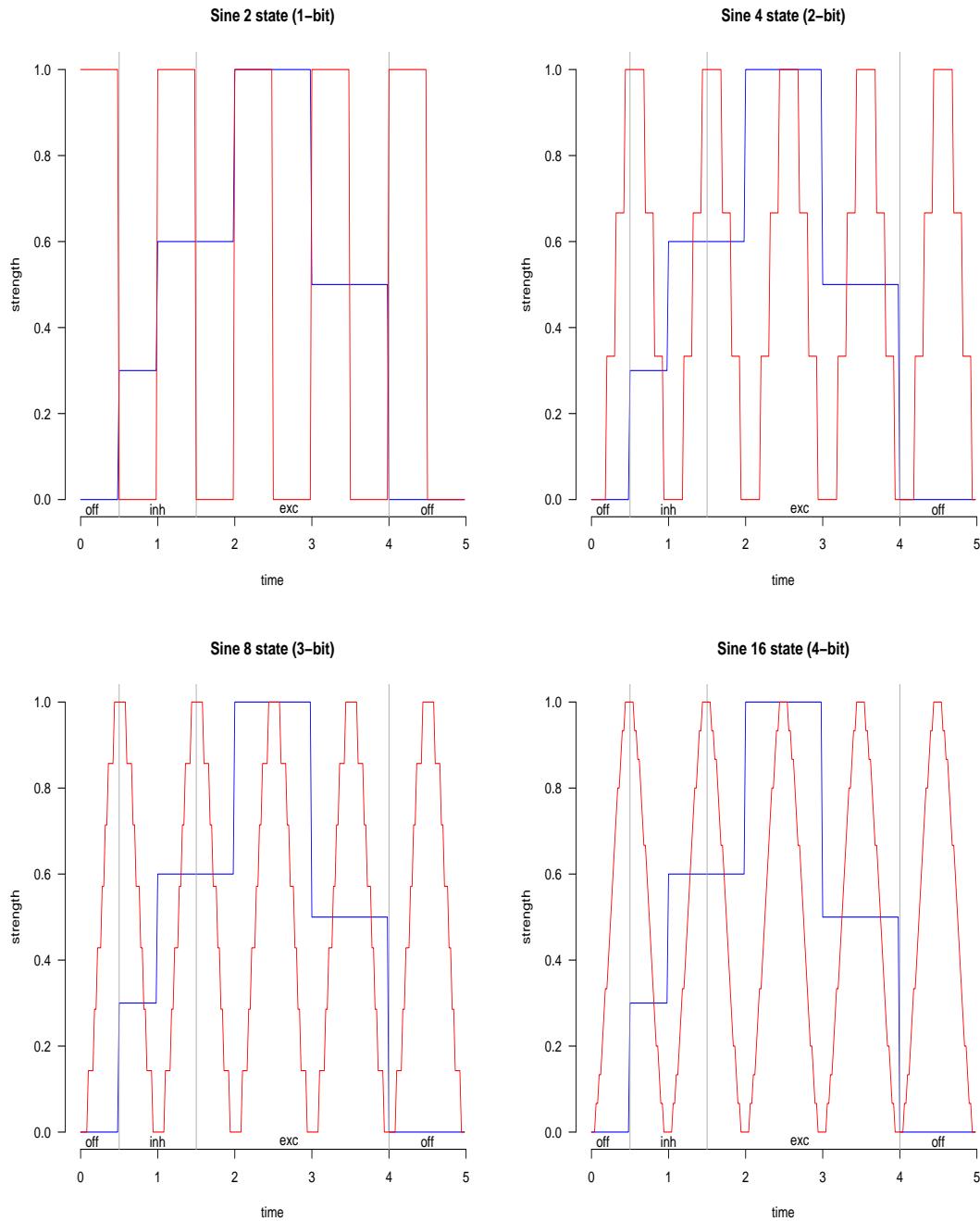
Gray vertical lines Transition times (0.5s, 1.5s, 4s) between the above stimulus states.

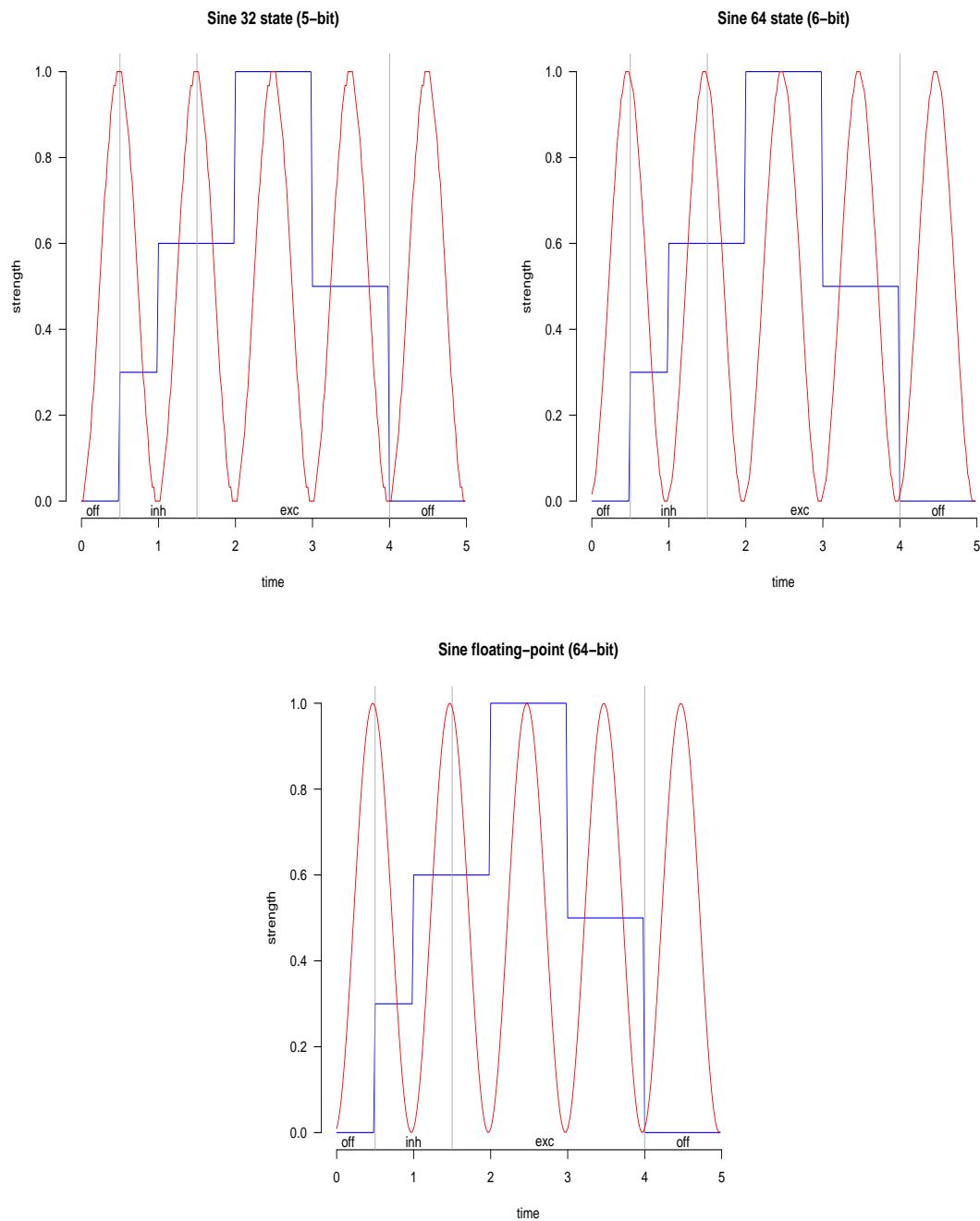
x-axis Time from 0s to 5s

y-axis Normalised signal strength. The only exception to this is for spiking neurons — spikes do not have a “strength”, so we only plot a short spike on the red line (neuron output) to indicate that one occurred.

9.1.2 Sine model

The sine model is described on page 60. These neurons generate a constant sine wave, regardless of the input stimulus. (It could be argued that this is not really a “neural” model, since all the node does is generate a sine wave, although Sims for example does refer to this kind of simple wave generator as a “neural node” [398].) This model has parameters amplitude and period. For these tests the amplitude was 1.0 and period was one second.

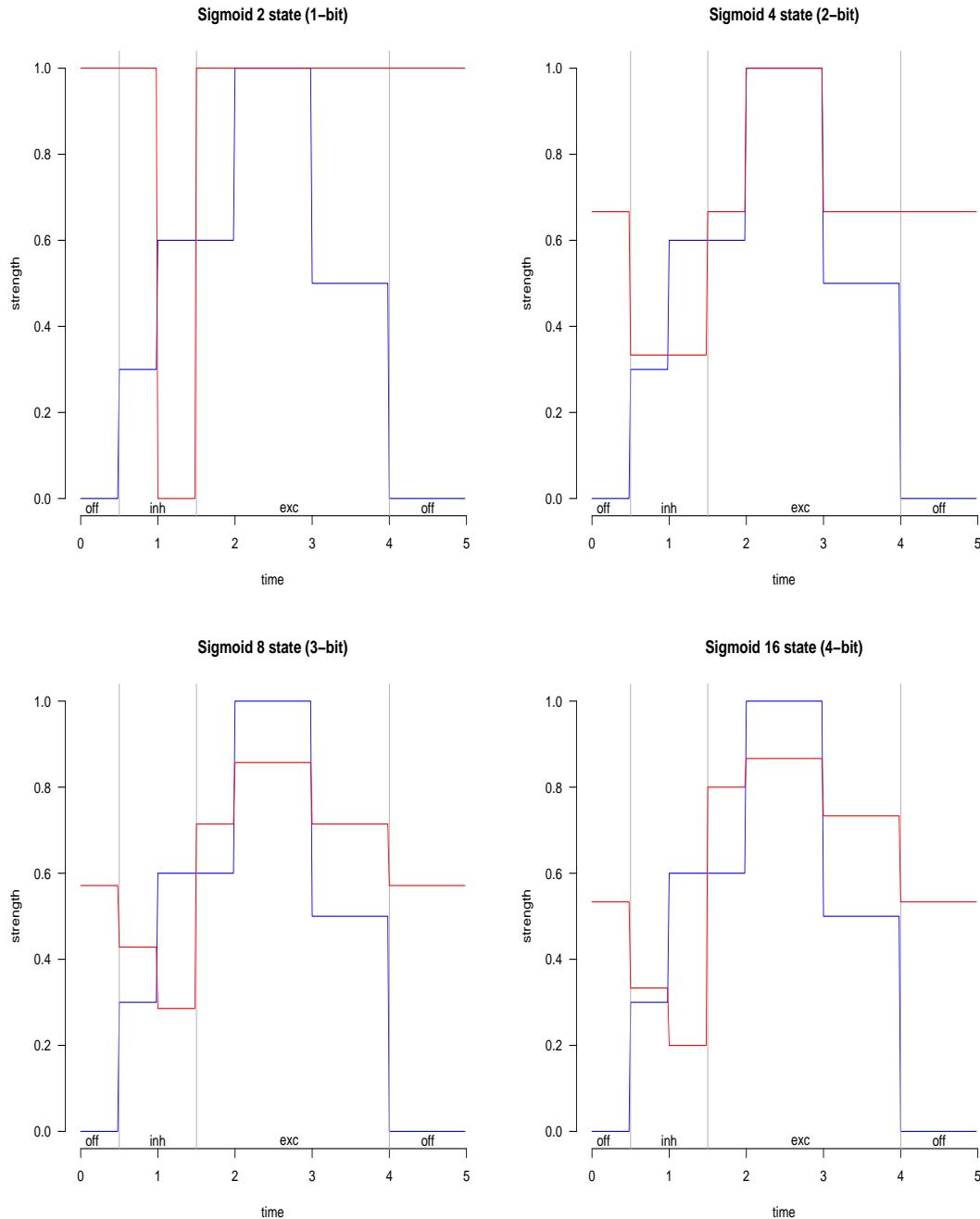


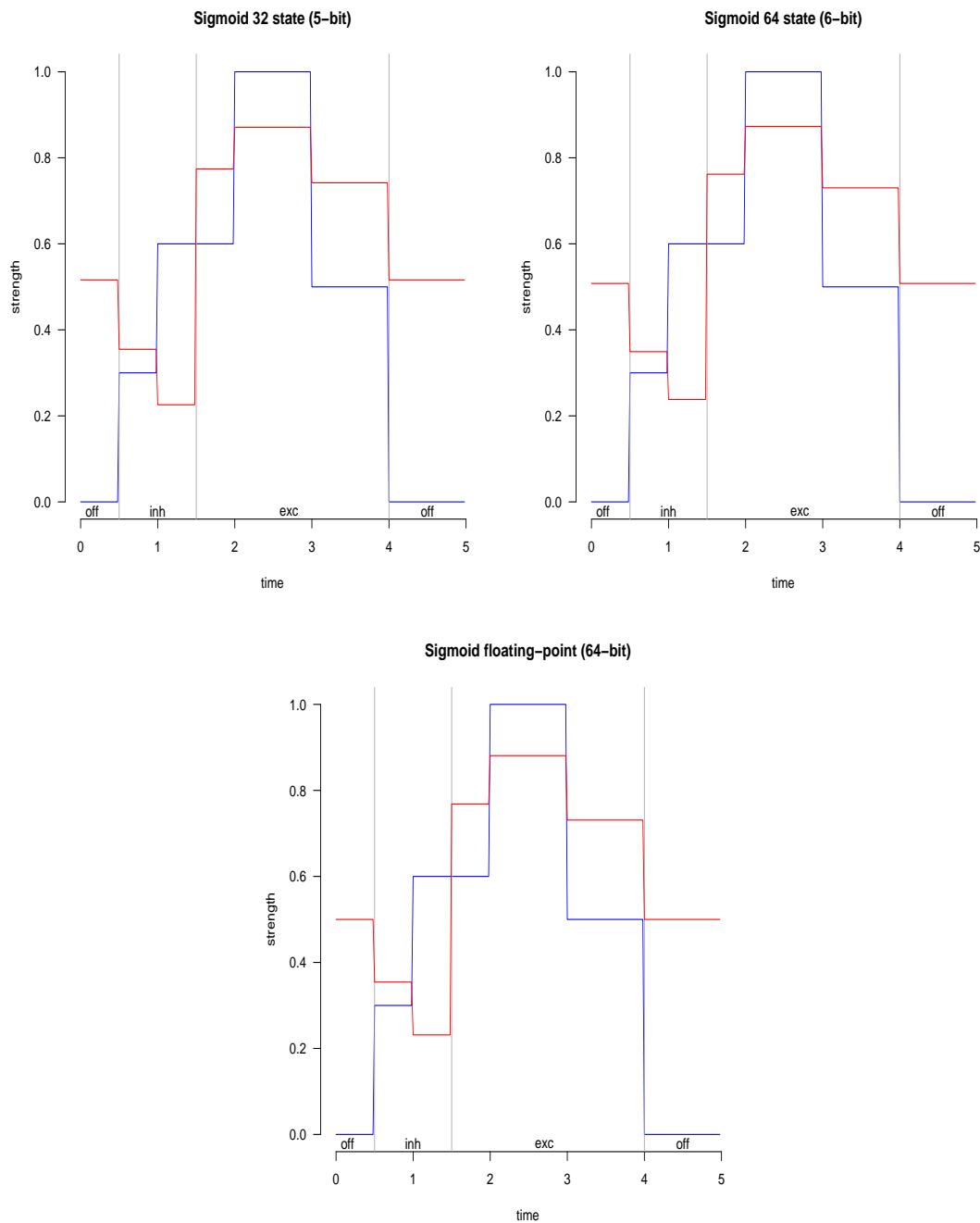


An oscillating output can clearly be seen in all of the plots. Note that the input signal is ignored as this model merely generates a plain sinewave. The floating-point implementation shows a smooth waveform, and it can be seen that this waveform becomes less smooth as the precision is reduced, to the point of being a binary oscillator at 1-bit precision.

9.1.3 Sigmoid model

The sigmoid model is described on page 57. The sigmoid output is a square waves, since the input stimulus is a square wave and sigmoid neurons have no internal state.





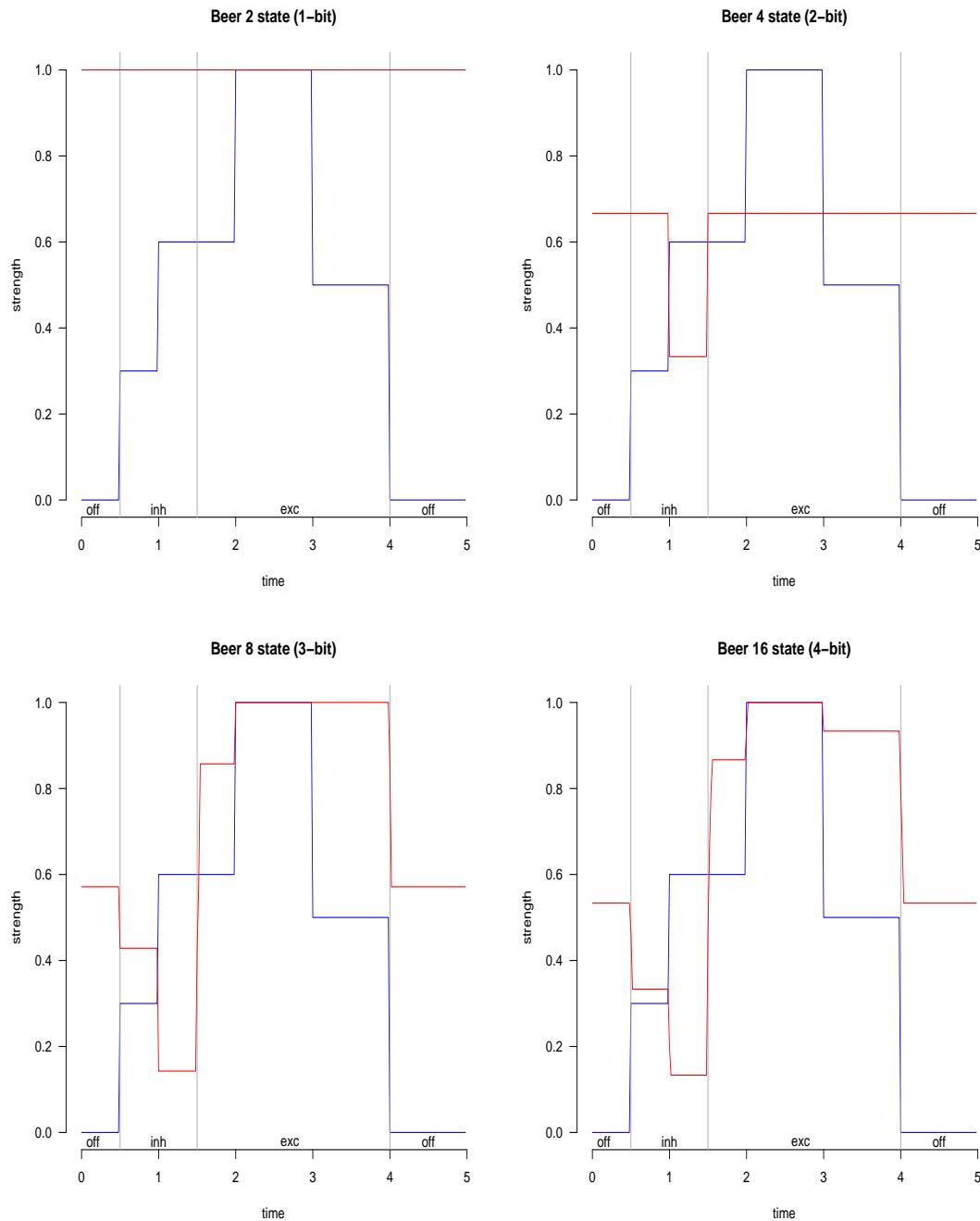
The sigmoid neuron has no internal state. Faced with the initial stimulus of 0 at 0s, the neuron generates an output of 0.5. As the input stimulus is increased at 0.5s, the output falls, since the connection is inhibitory. When the connection is changed to be excitatory at 1.5s, the output goes high, and then follows the input signal. When the input signal returns to 0 at 4s, the output returns to the initial value of 0.5 immediately, as there is no internal state.

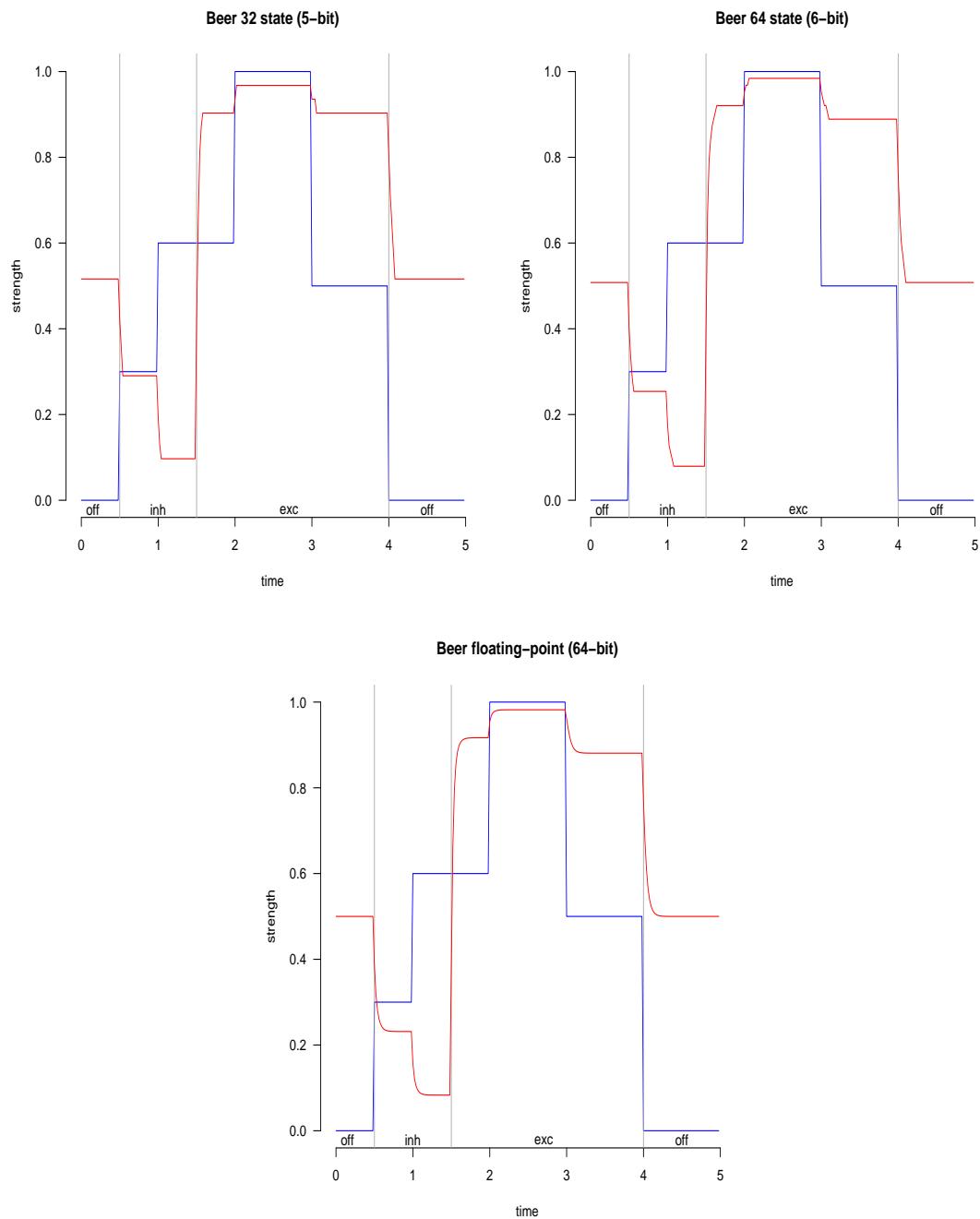
The quantised models show a great degree of similarity to the floating-point model. Down to 8 state (3-bit) the response is the same shape, and almost same magnitude.

The 4 state (2-bit) model loses resolution and the output signal only takes 3 different values (as opposed to 5 values for the 3-bit model). Despite this, it can be seen that the waveform is similar to those of models with greater precision. The binary state model does preserve the fall in signal between 1s and 1.5s, but otherwise fails to accurately approximate the detailed shape of the higher precision models.

9.1.4 Beer model

Beer's "continuous time recurrent neural network" model is described on page 58. The Beer model exponentially rises or decays towards some equilibrium level depending on the stimulus. The neuron's *bias* was 0 and the *adaptation rate* was 0.05.



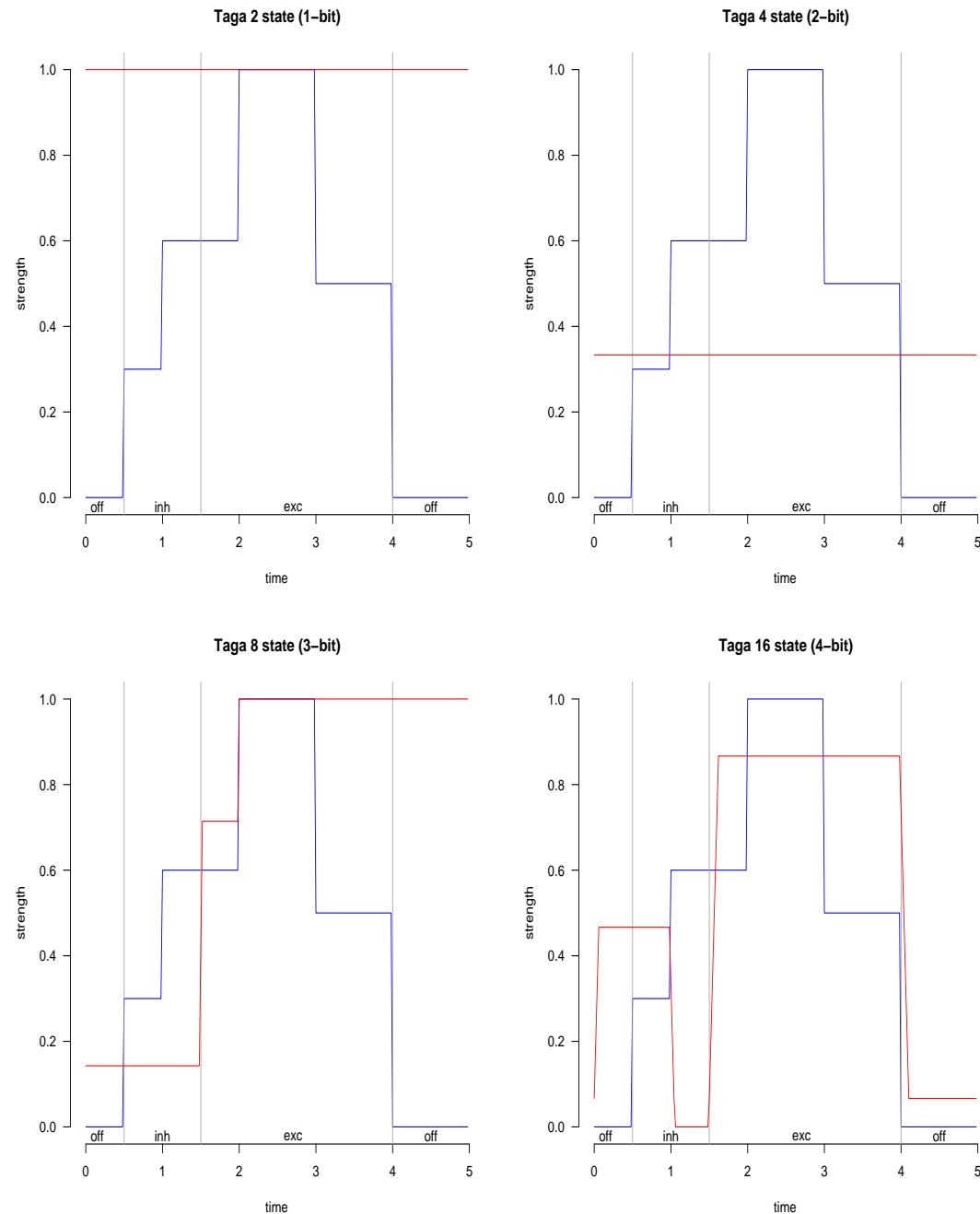


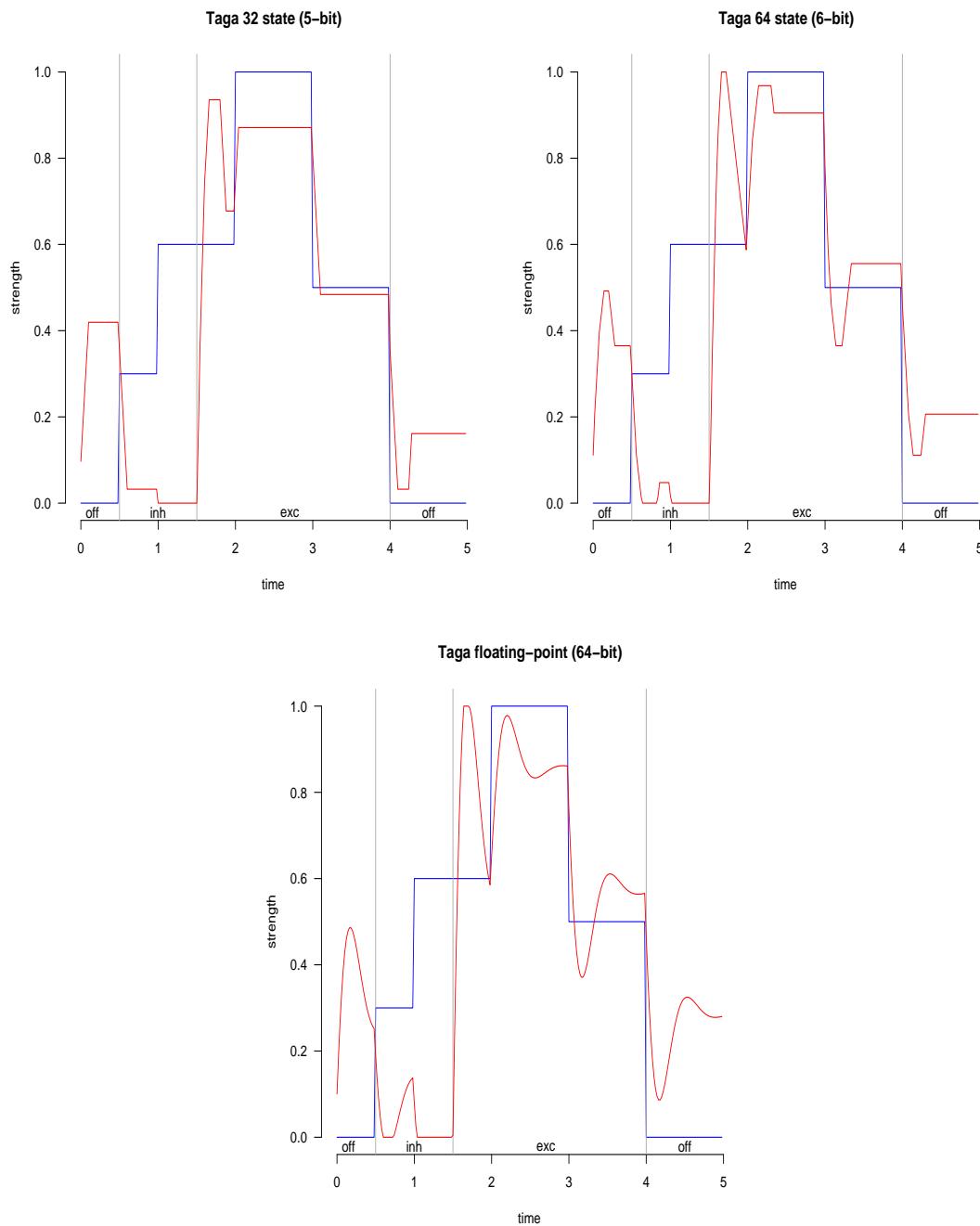
The floating-point model starts with an initial output of 0.5 at 0s where the input signal is 0. The input signal is initially inhibitory, so when it rises at 0.5s the output goes down. Note that the output signal shows signs of being affected by an internal state — the fall is not immediate, which would generate a square wave, but curved, showing a rapid initial decrease and levelling off. The same occurs at 1s. At 1.5s the input signal is switched to excitatory, and the output signal rapidly rises, and again at 2s. At 4s the “leaky” behaviour of the neuron can be seen, when the input drops to 0 and the output quickly falls to 0.5 as a result of the fall in internal state.

The effects of reducing the arithmetic precision is already observable going from floating-point to 6-bit, where the smooth curves of the floating-point signal are replaced by a linear approximation. The basic shape of the waveform, with 6 distinct output levels, is preserved down to the 4-bit model. The 3-bit waveform loses only one stable output level, but beyond that the waveform is essentially lost.

9.1.5 Taga model

Taga's neuron model is described on page 58. Two linked differential equations should generate more complex behaviour than the sigmoid model or Beer's CTRNN model.





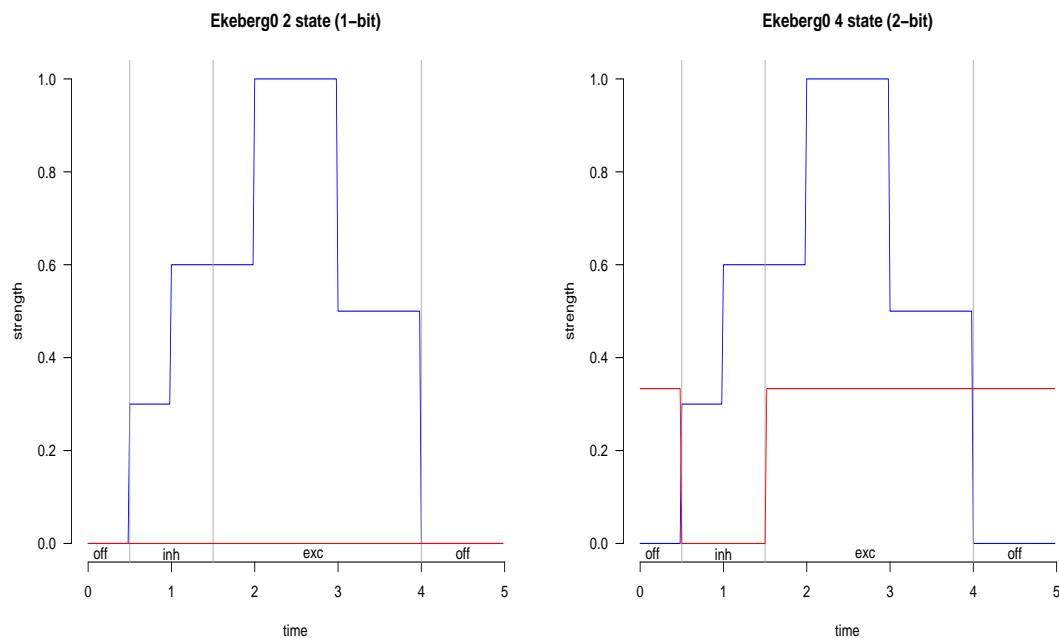
The floating-point model here shows a complex output waveform. The output initially falls in the absence of input at 0s. The fall is accelerated at 0.5s by the increasing inhibitory input. When the input connection switches to excitatory at 1.5s the output rapidly rises. It can be seen that the output “overshoots” its stable value, and then falls towards it — the same thing happens when the input signal changes at 2s, 3s, and 4s. For example, just after 3s the signal falls, climbs, and falls again to reach a level output. This process takes almost one second.

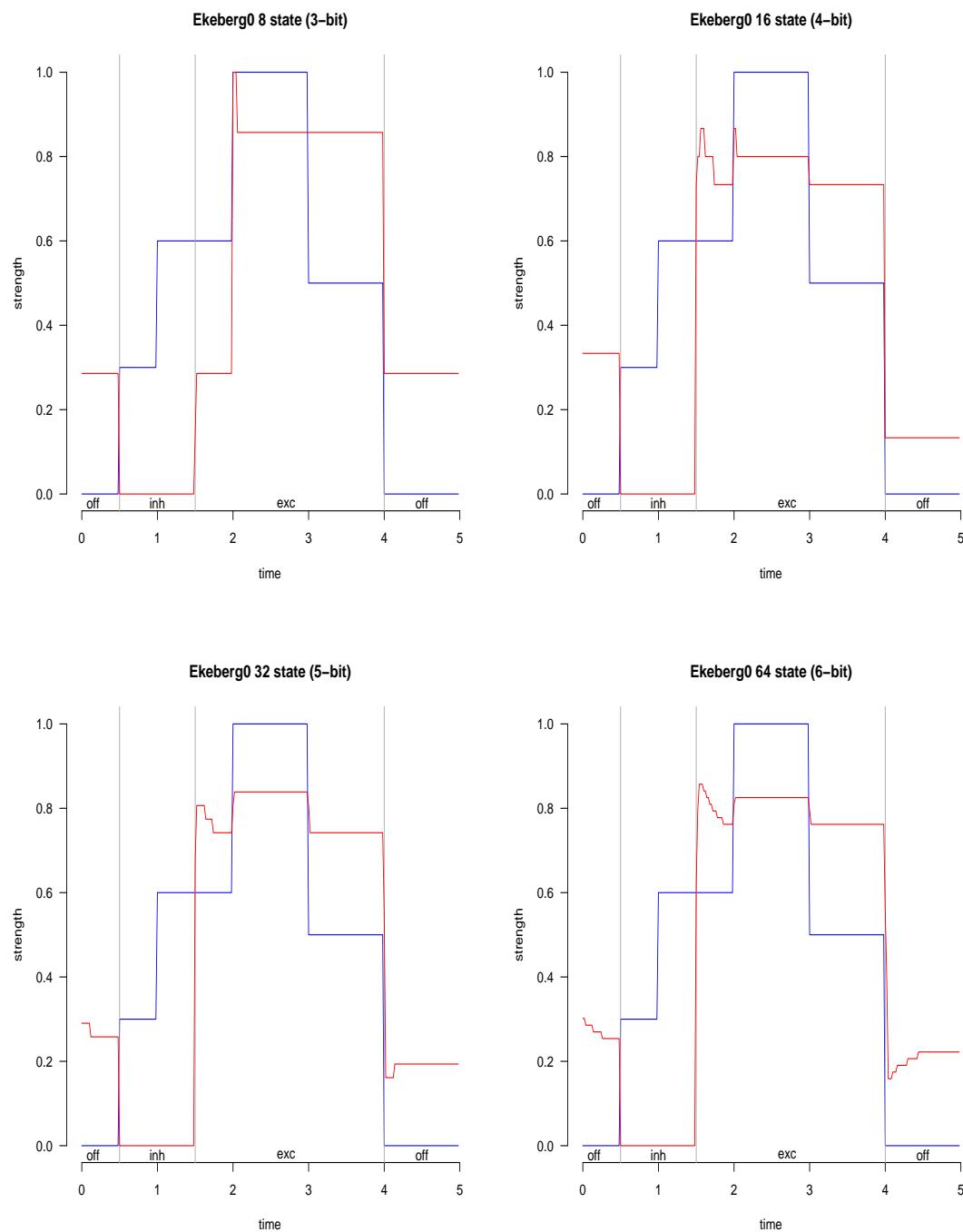
The floating-point model shows the greatest resolution. The output of the 6-bit model already shows signs of linearisation, but the waveform is mostly the same. The “overshooting” of the stable values is observable, but it only occurs once for each input change — the small secondary overshoot that is visible with the floating-point model (e.g. from 3.5s to 3.8s), is not visible anymore. The 5-bit model retains most of the peaks and troughs. The basic shape of the waveform is recognisable down to 4-bit precision. At 3-bit the waveform is not the same, and the output with 2-bit and 1-bit models is constant and not responding to the varying input stimulus at all.

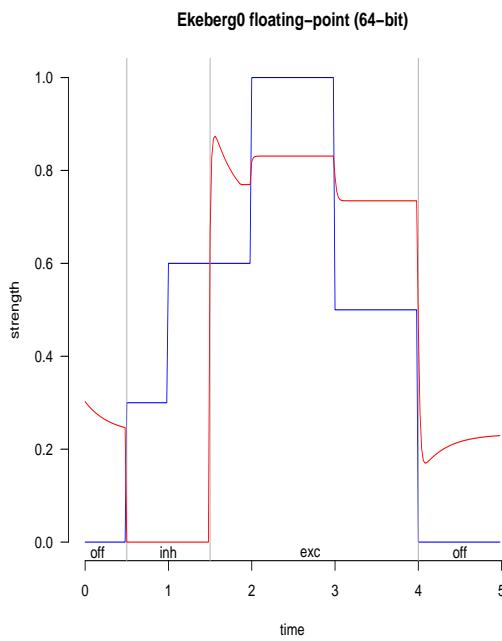
9.1.6 Ekeberg model

Ekeberg's neuron model is described on page 59. There are four versions of the Ekeberg model, corresponding to the four parameter sets provided by analysis of biological neurons.

In this model, connections between neurons have no polarity. Individual neurons are either wholly excitatory or inhibitory, so after 1.5 seconds the stimulus neuron is switched from inhibition to excitation, generating the same effect as inverting the connection weight does for the other tests.

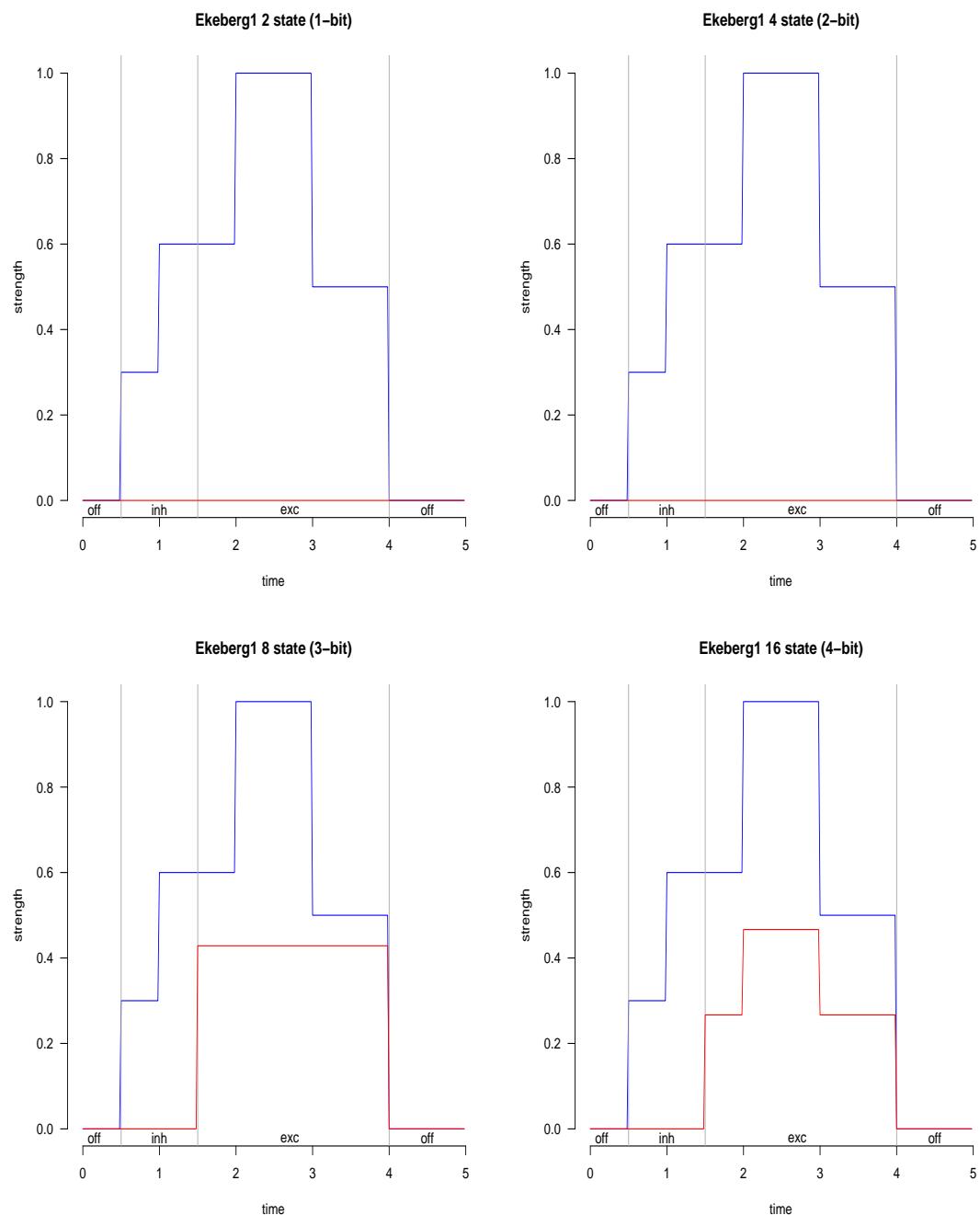


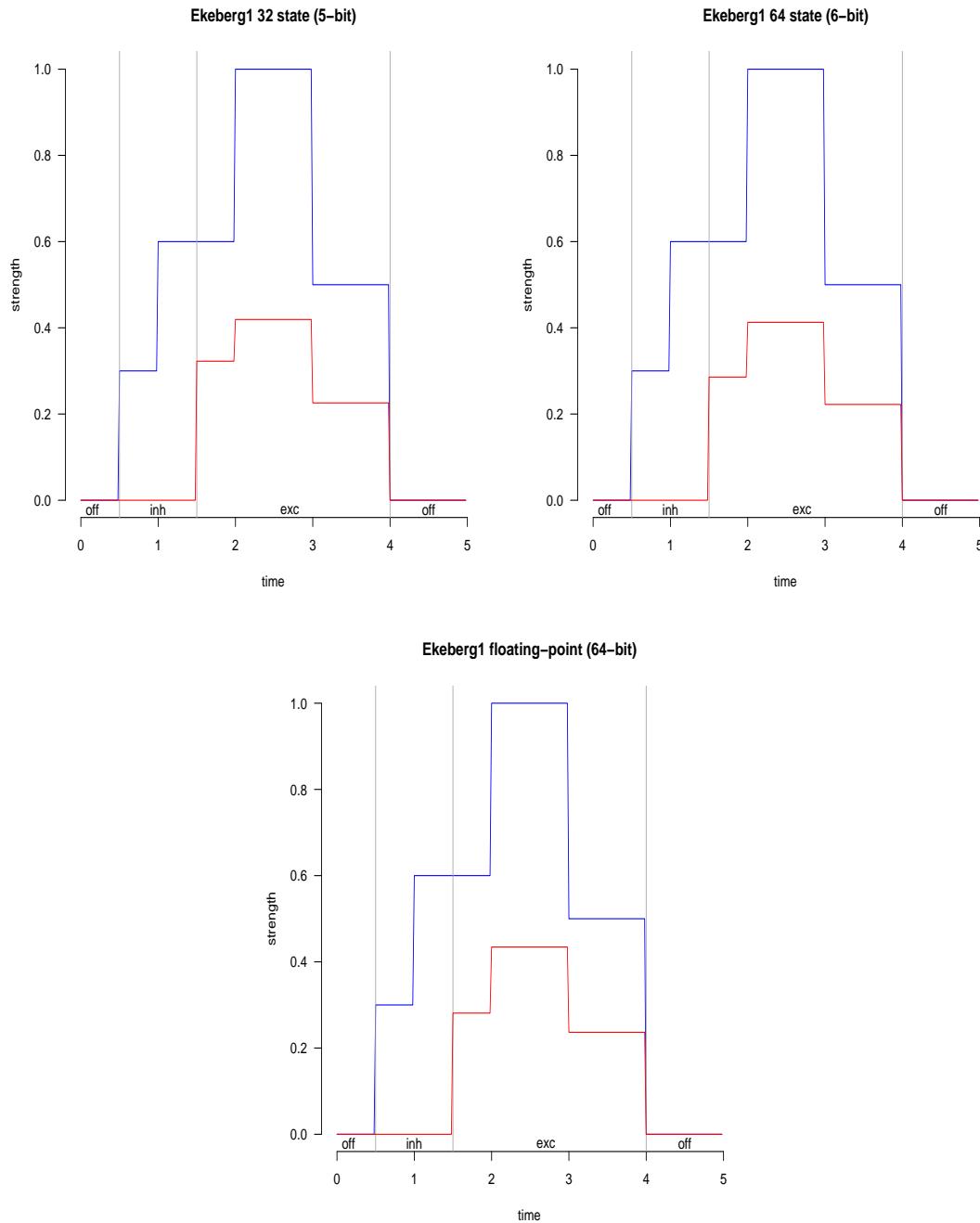




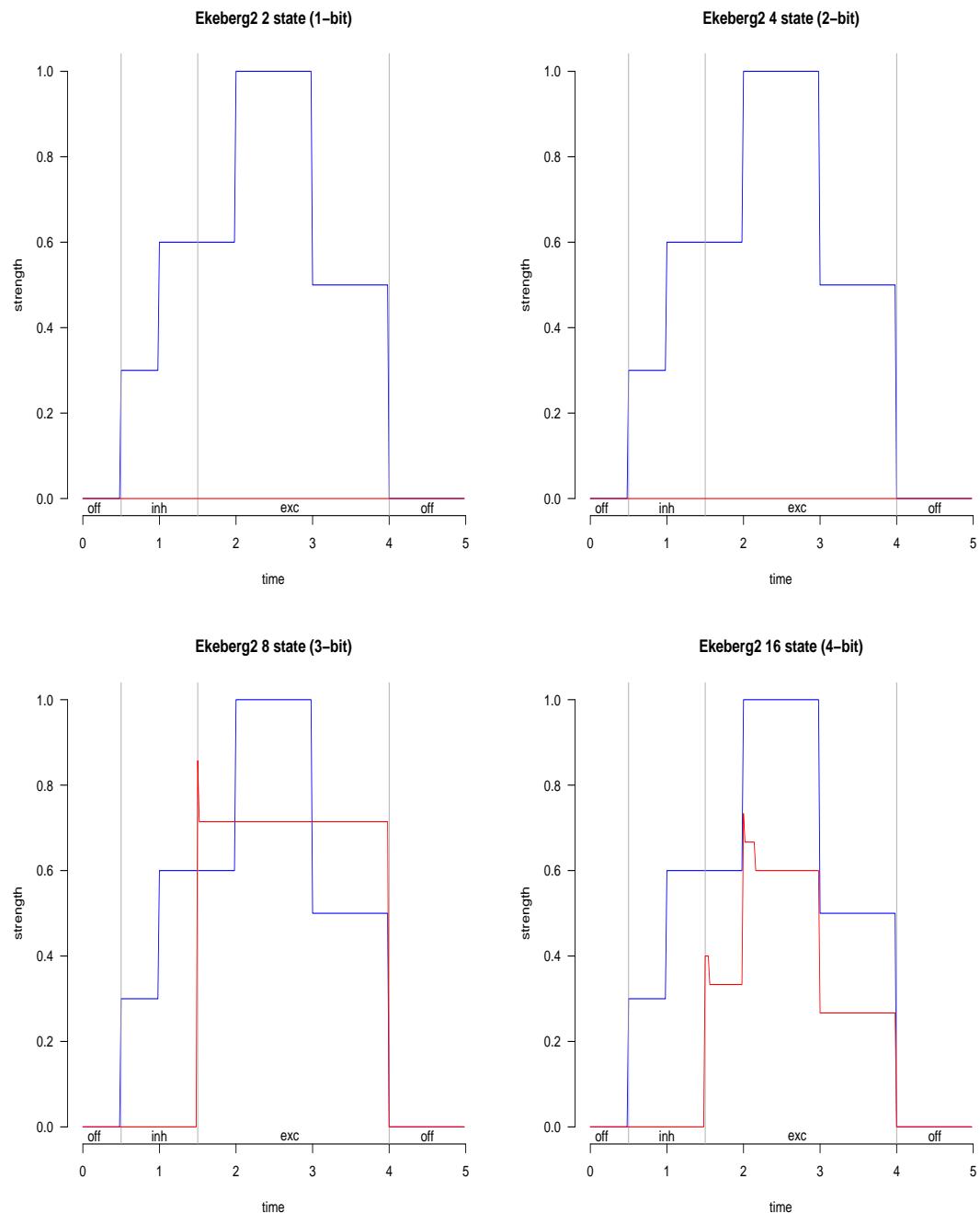
For each change in the input signal there is a corresponding change in the output signal. The initial inhibition causes the output to fall to 0 at 0.5s. After the stimulus changes from inhibition to excitation the output rapidly increases. The response output after this follows the input signal in either rising or falling, although there are some signs of overshooting followed by correction, e.g. at 4s. Despite the claims of more advanced behaviours being possible, the neuron displays no signs of oscillation or other complex movements.

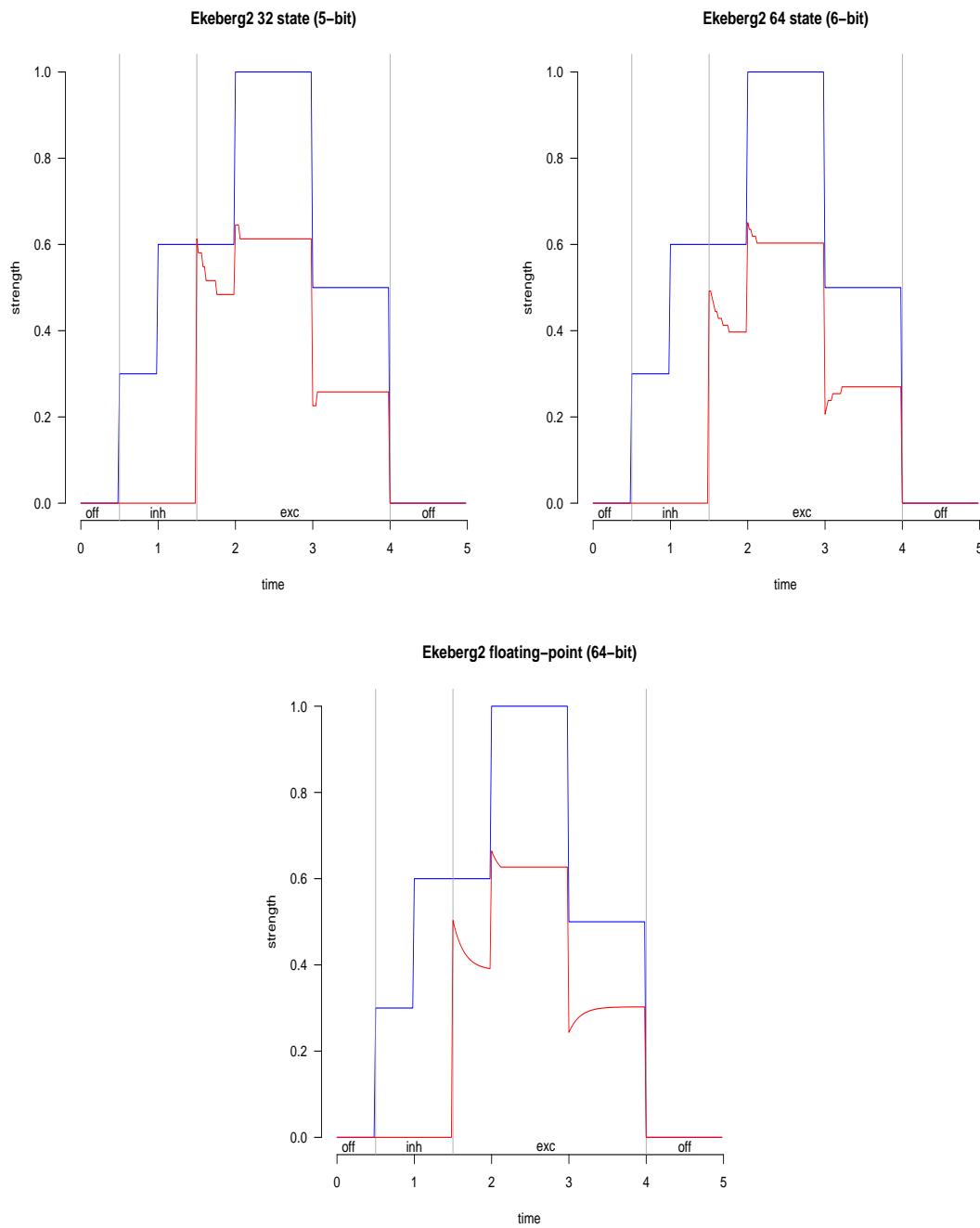
The waveforms of the floating-point model, 5-bit model and 4-bit model are all similar. At 4-bits some of the small variances disappear, but a new spike is introduced at 2s. The 3-bit output waveform has a similar overall shape. Beyond that, at 2-bit and 1-bit, the output waveforms are not recognisable as the originals.





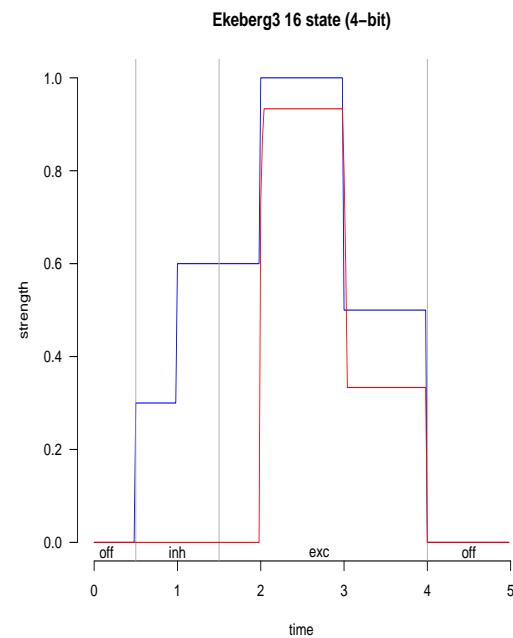
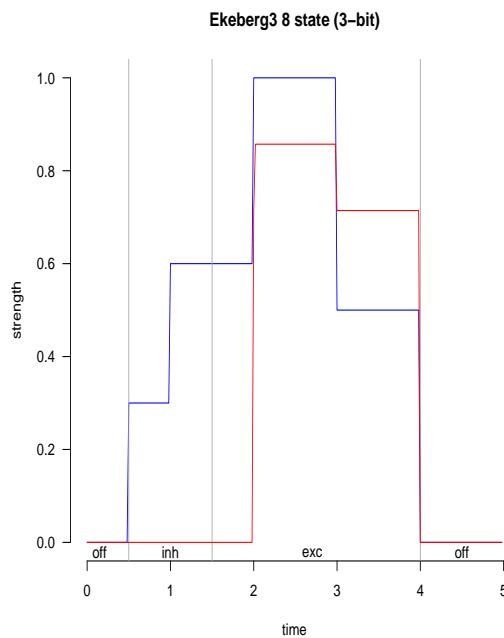
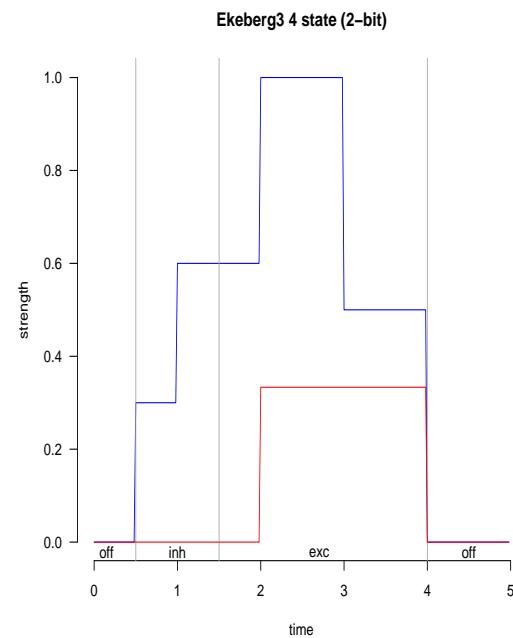
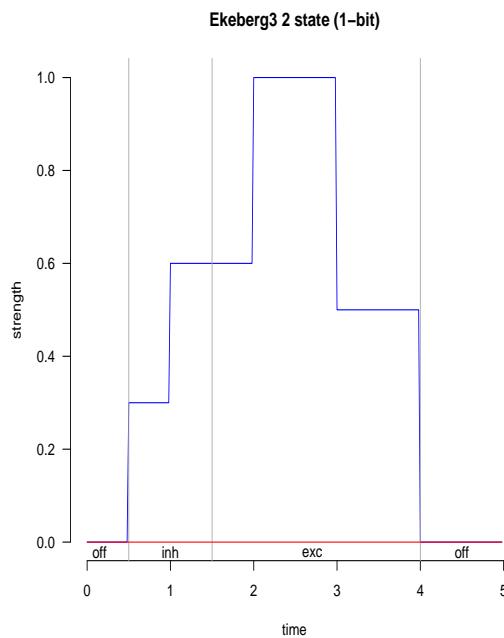
The output here has sharp edge transitions. There are few levels (only 4) even in the floating-point model. The output waveform is preserved down to 4-bit precision, but beyond that is clearly different to the original. For 1-bit and 2-bit the output is constant. The lack of curves or any other sign of complex behaviour suggest that this model is not making use of the complex internal state.

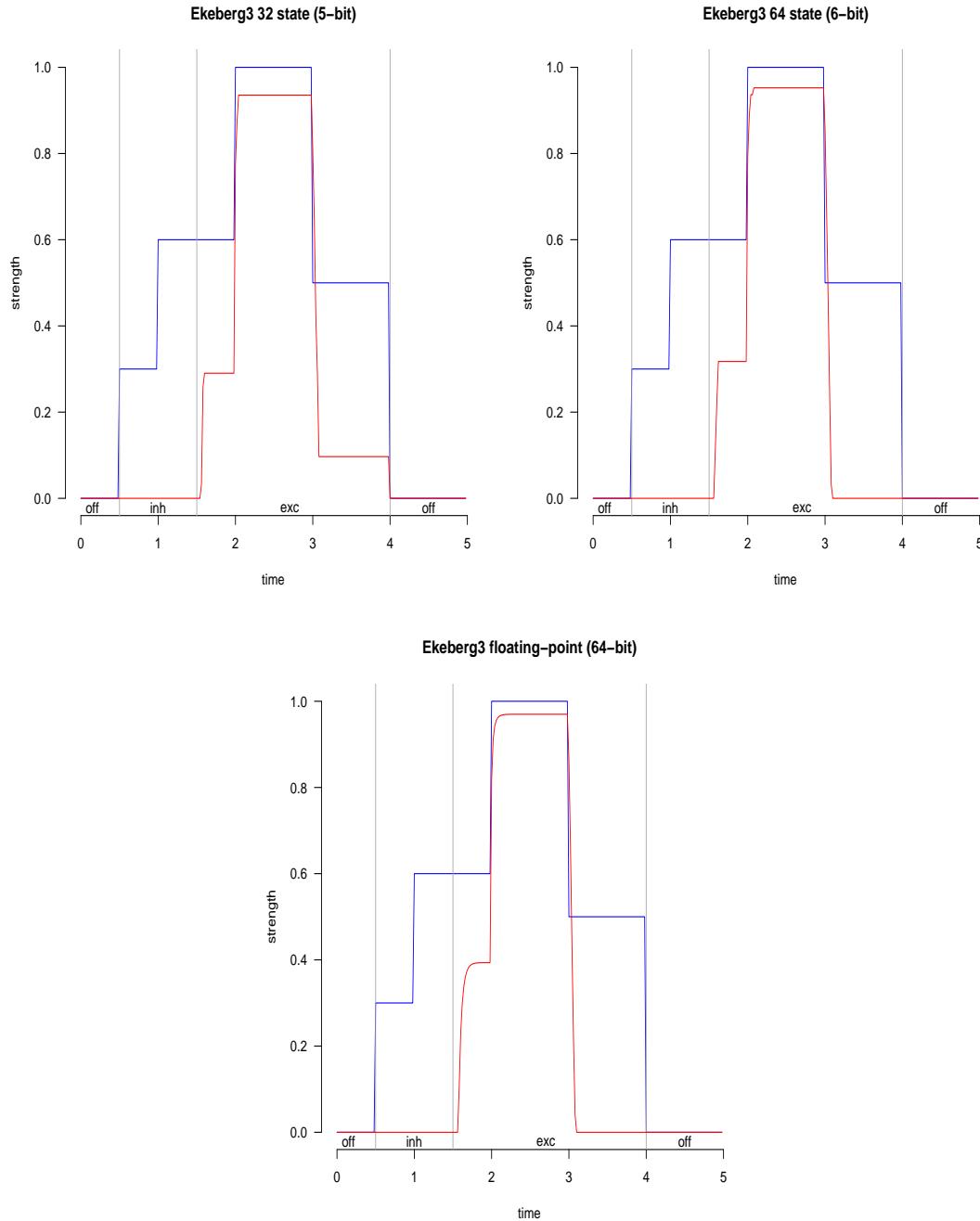




The output here remains constant 0 in the face of an inhibitory input signal. At 1.5s the signal becomes excitatory, and the output responds by rising. The output adjusts in response to changes in the input signal, it does “overshoot” the value it is tending towards but quickly stabilises. There are no signs of oscillation or any other complex behaviour.

The shape of the waveform is preserved down to a precision of 4-bit. At 3-bit the basic shape is there, but several stable levels have been lost. The 2-bit and 1-bit model both have a constant output of 0.





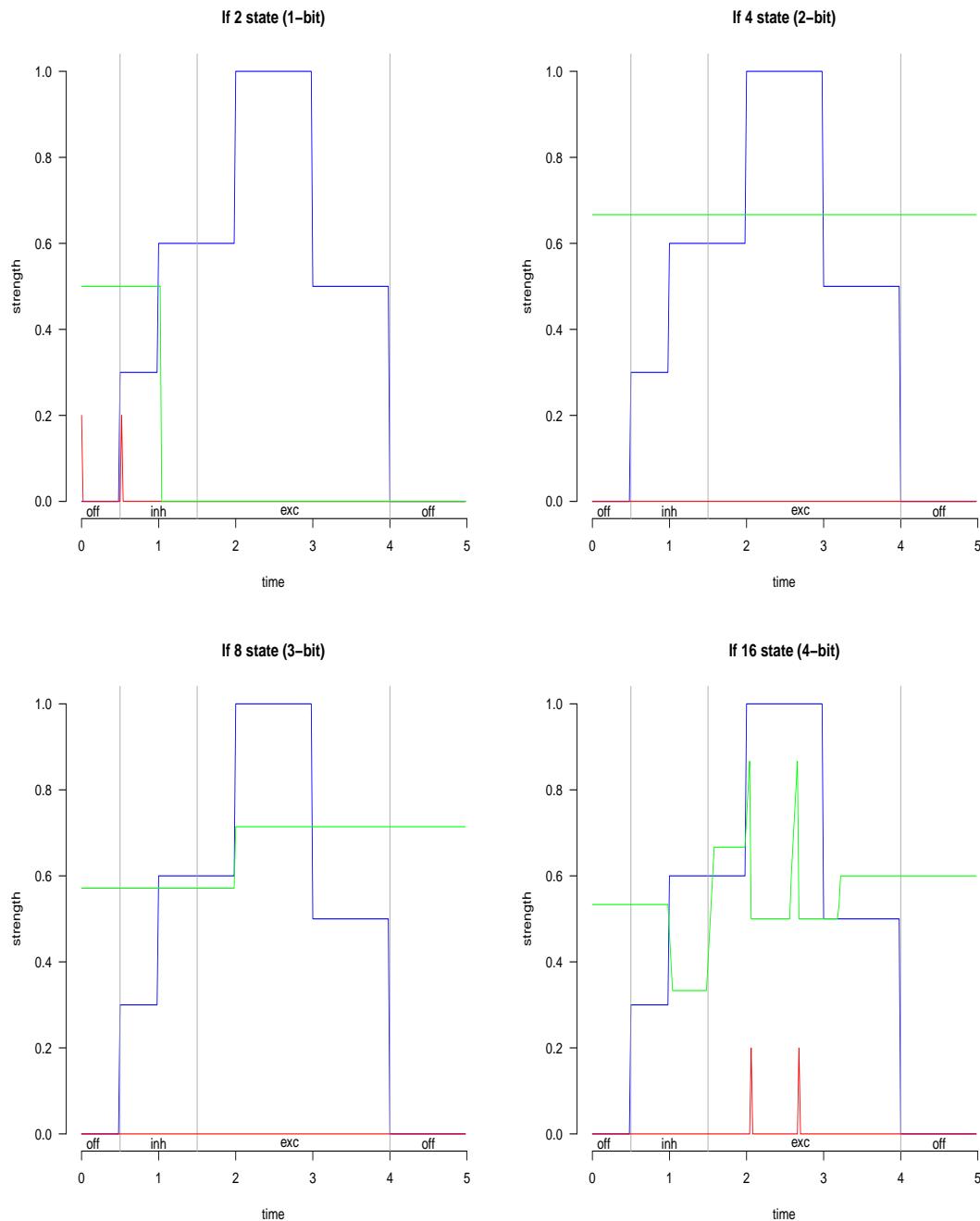
While the input is inhibitory the output is completely suppressed (forced to 0). At 1.5s the input changes to excitatory, and the output rises in response. The output follows the input signal, and at 3s falls back to 0. There are very few stable levels of output, and the response appears quite simple. There are no signs of oscillation or any other complex behaviour.

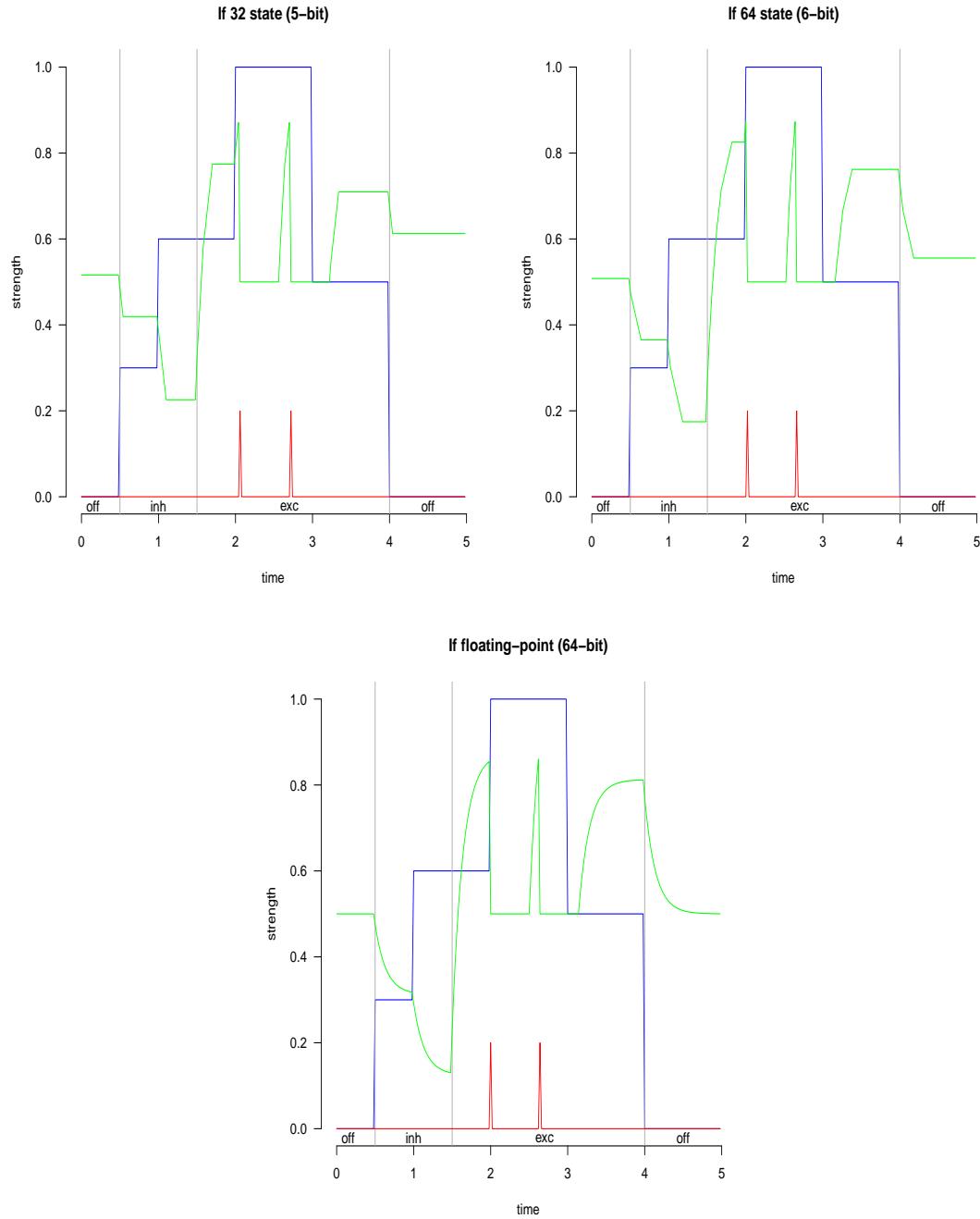
The floating-point output waveform appears simple to preserve as it has only 3 stable levels. However, the 6-bit model is the only one that accurately reproduces the floating-point output. At 5-bit precision a new ‘lump’ is visible from 3s to 4s. As the

precision is reduced to 4-bit and 3-bit this value at 3s to 4s grows, making the waveform different from the original. At 2-bit precision the waveform is unrecognisable as the original, and at 1-bit the output is just constant 0.

9.1.7 Integrate-and-fire model

The integrate-and-fire neuron model is described on page 53. In the graphs following, the internal state is plotted as a green line. The red line along the bottom shows outgoing spikes, which are generated when the internal state exceeds some threshold. After each spike the internal activity falls. In these tests the neuron's *bias* was set to 3 and the *adaptation rate* to 0.15.





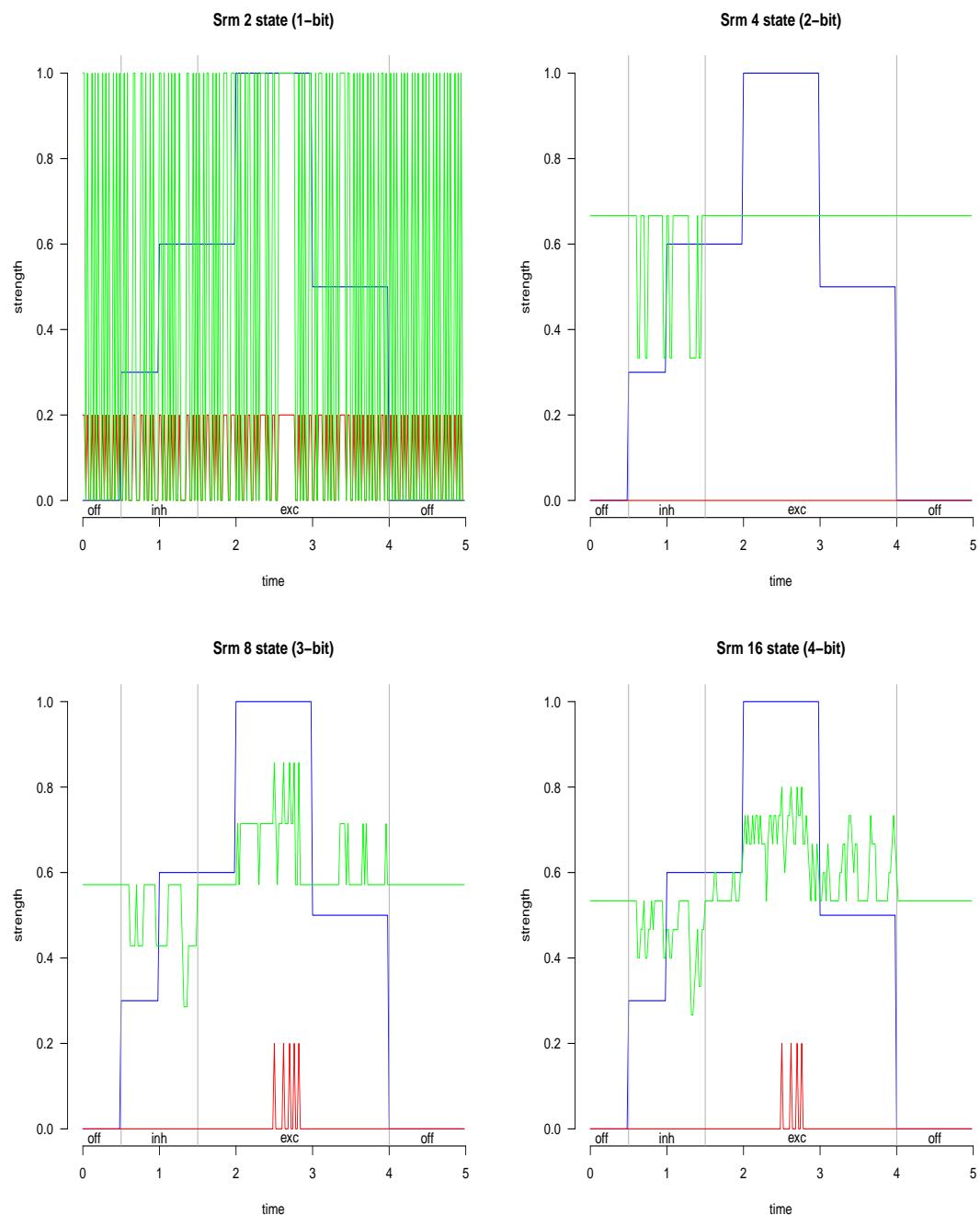
The floating-point version here shows the internal state falling in response to activity on the inhibitory input. After 1.5s the inhibition turns to excitation, and the internal state rapidly rises, triggering an output spike. The internal state is reset to 0 and the neuron enters a refractory period. Once this times out, the internal activity rises again, generating another output spike. During the subsequent refractory period the input signal falls to 0.5 at 3s. Now the neuron again leaves its refractory period and internal activity rises, but slower than previously due to the lower input stimulus. The internal state stabilises just below the threshold for firing, so there is no spike this time. At 4s

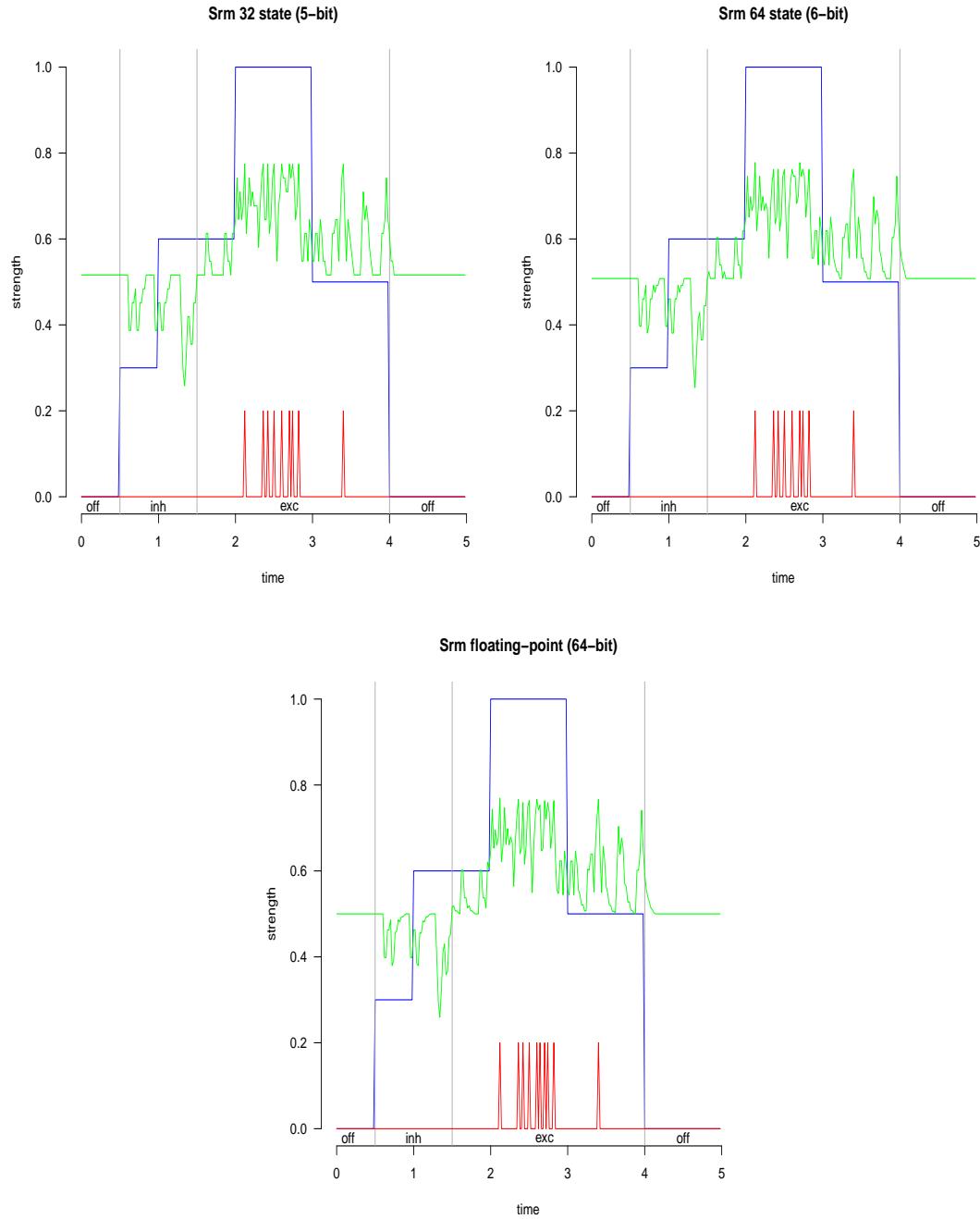
the input signal drops to 0 and the internal state falls back to its resting potential.

The interesting thing to analyse about this spiking model versus the previous continuous models is whether the outgoing spikes are preserved as the arithmetic precision is reduced, as these are the only events visible externally. The floating-point model generates two spikes. The models with 6-bit, 5-bit and 4-bit precision preserve the internal state waveform well, and the two outgoing spikes are preserved. At lower precision the approximation breaks down, with a completely different internal state waveform, and no spikes for the 3-bit and 2-bit models. The 1-bit model generates two spikes immediately, which is presumably an artifact of the binary quantisation of the input signal, and shows no activity at all after 1s.

9.1.8 Spike response model

The spike response model is described on page 55. The stimulus applied to the SRM model consists of spikes randomly drawn from a Poisson distribution with probability proportional to stimulus level. SRM, as described in section 3.4.4.2, has a spike effect over 20ms. Since the physics simulator's integration time step is only 20ms, the implementation of this neuron model here uses ϵ and η functions stretched to cover 200ms instead. The neuron's *firing threshold* parameter was set to 2.0 here (as previously stated, the values of all variables are normalised in these plots).





The internal state appears to have some complex behaviour, and shows some oscillation even while the input stimulus is held constant. While the input is inhibitory and active from 0.5s to 1.5s the state shows some tendency downwards away from the resting potential of 0.5. After 1.5s the input becomes excitatory and the internal state rises above the resting potential. It is not until 2s that the input activity becomes large enough to cause the neuron to fire and generate an output spike. The floating-point model generates 10 output spikes in total. There is a lone spike after 3s when the input activity has fallen. The effect of the fall in input activity can be seen - the activity of

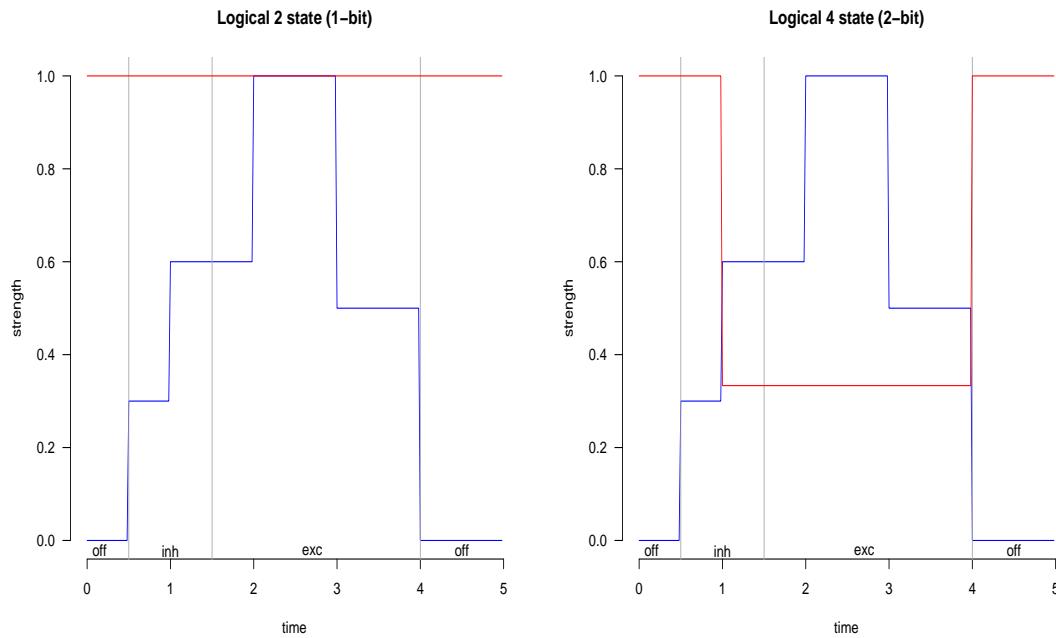
the internal state from 2s to 3s is clearly greater than from 3s to 4s. At 4s the stimulus falls to 0 and the neuron state quickly falls to its resting potential.

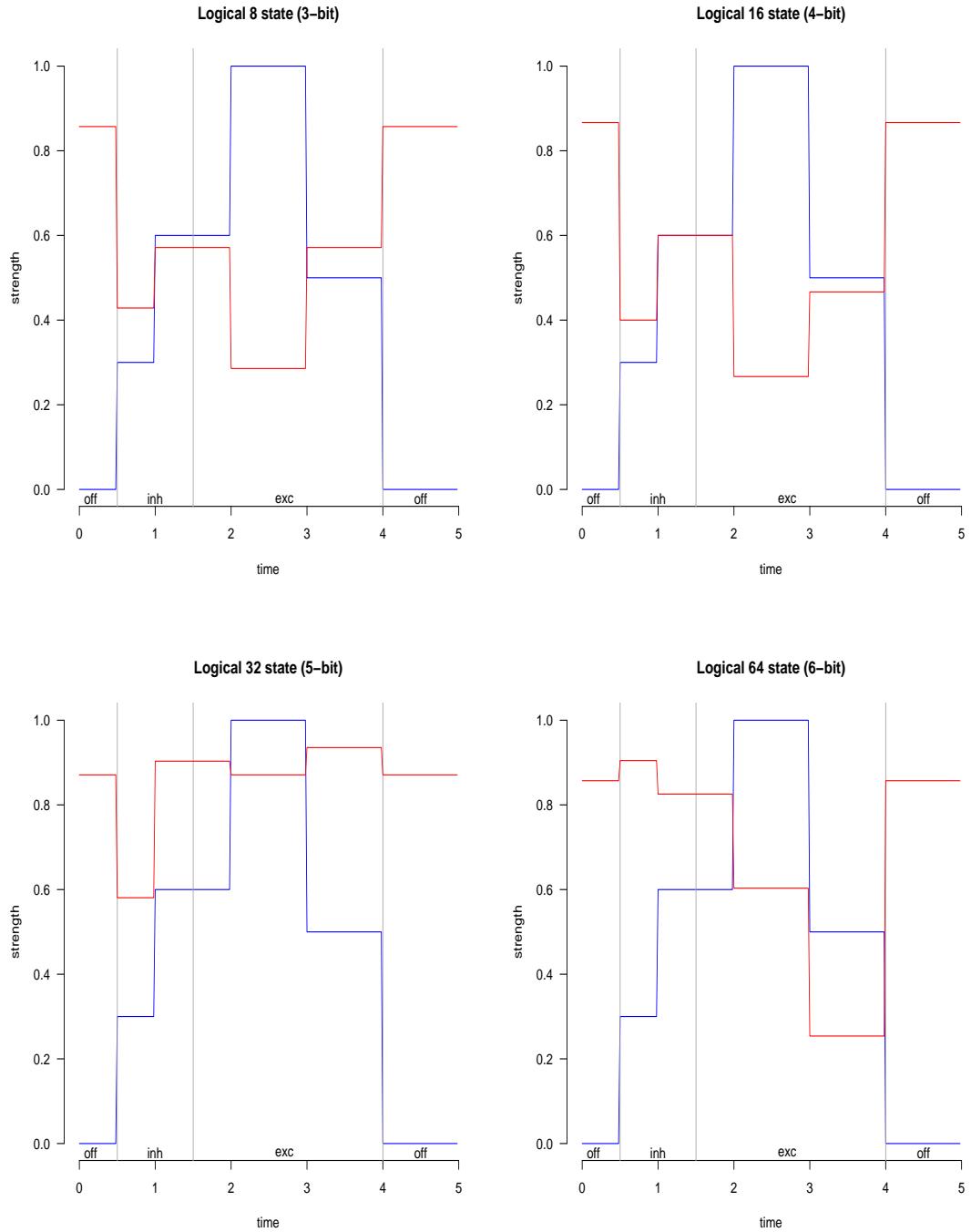
Reducing the precision from floating-point to 6-bit causes one less spike to be generated (9 spikes total), although the internal state waveform is substantially similar. The 5-bit model also generates 9 spikes, with a similar activity. At 4-bit precision, again the internal state waveform appears similar, but now only 4 spikes are generated. Most of the fine waveform details have disappeared at 3-bit precision, but there are 5 output spikes. At 2-bit precision there are no output spikes. The 1-bit output spikes repeatedly due to the internal state oscillating.

9.1.9 Logical model

The logical model uses a lookup table with random valued entries. Due to this, and unlike the other models, we should not expect some obvious similarity between the different precisions. There is no continuous version of the logical model, since by definition it can only be discrete valued.

Note that the switch from inhibition to excitation makes no difference here. The input value is merely transformed into an address that is used to index into a lookup table to find the output value.





There is not much to say about these results for the logical model, and they are included only for completeness. As expected, there is not necessarily any correlation between different precision models — there is no correlation between the values held in the lookup table, because it was randomly initialised. There would only be some correlation if the same lookup table was used, initialised in the same way, using the same random seed.

9.2 Cluster performance

In order to quantify how many simulations for the purpose of fitness evaluation could be carried out testing was done on a 64-client Linux cluster. Each host had a 1.8GHz Pentium 4 processor. The client software is small and not bounded by memory or disk space. There is little network activity — the representation of a complete genotype requires around 15KB of storage, and the time required to transfer such a small amount of data across a modern high-speed network is negligible compared to the time required to carry out a single fitness evaluation. In theory high loads on the central database server would affect results; in this case, the server was checked to make sure there were no other CPU intensive tasks running.

Population size and number of clients are linked in the context being discussed (run speed), because all individuals must be evaluated before progressing to the next generation. If there are n hosts, and fewer than n individuals, then we can complete all evaluations in parallel (assuming that evaluations take a similar time) in one cycle. With n hosts, we can only process n simulations at a time, so obviously it will take (roughly) twice as long for $2n$ simulations, three times as long for $3n$ simulations, and so on. Having a small population relative to the number of hosts will mean that some hosts are doing nothing a lot of the time - e.g. if we have 50 hosts, but only 5 individuals, then we will only run 5 simulations in parallel, so 45 of the hosts will be doing nothing. The aim is to maximise usage of the available computational power.

Two simple parallel scheduling algorithms were implemented, one that assigns tasks linearly to hosts in order to avoid unnecessary repetition, and another that allowed tasks to be assigned randomly. If a host was already highly loaded, completion of the task would be delayed. With the first scheduler this would delay progress of the genetic algorithm, but with the second another host could be assigned the same task, allowing hosts to compete against each other.

Each evaluation actually carries out three separate simulations with an identical configuration but different random seed, which means the initial state of the neural networks will be different. The lowest fitness score is returned as the final result. This is done to prevent freak high scoring simulation runs from producing individuals which will dominate the population, as we seek consistent performance from an individual, not a single instance of greatness or some fluke numerical inaccuracy in the physics simulation. The precise evaluations being done were simulations of evolved locomoting creatures.

The next generation will not be created until all of the individuals in the present generation have been evaluated. This means that any single evaluation that takes a long time to complete could hold up the evaluation of the population as a whole. Evaluations would take a long time if the simulation was particularly complex, or if it had been assigned to a slow PC (due to either the hardware being slow, or the system running other processes).

If the number of individuals in the population exceeds the number of PCs available, some PCs will be required to evaluate more than one individual. These simulations will be carried out in sequence, which necessarily lengthens the time to completion. A similar problem occurs when a simulation is assigned to a slow PC — quite often the dataset of the individual can be duplicated and dispatched to another free PC (which has already completed processing of its first individual evaluation), and this new PC will return an evaluated fitness faster than the slow PC that the dataset was originally assigned to.

Results are shown in figure 9.2. As can be seen, linear scheduling performed better initially due to lack of duplication. As the number of hosts was increased above about 45 performance began to decline, and at around 56 hosts the random scheduler became the better performer. This suggested that, on this particular cluster, and for populations of between 30 and 50 (a number often used in evolutionary experiments), linear scheduling would enable the experimental runs to be performed more quickly, as the whole population could usually be evaluated simultaneously in a single cycle.

Upon investigation it turned out that a small number of the PCs in the cluster would already have an existing workload from other users (the cluster is shared, but the processes of any user are supposed to have exclusive access to a PC when run) that for some reason had not been killed when the user terminated (daemon processes without a controlling *tty* were one class of process that was confirmed to not be terminated). Hence limiting the assignment of each dataset to a single PC caused performance degradation when any of the PCs happened to already be running another user's processes, a situation that was not supposed to happen. Although eliminating this particular problem would fix the issue here, a more robust algorithm also has advantages in other use cases where individual PCs may, for reasons unknown, run slowly (e.g. research on hard disk drive performance in computational clusters has shown that some drives exhibit performance far below that of supposedly identical drives from the same manufacturer, but that this poor performance is unpredictable as the drives do not generate any error or failure signals [333]).

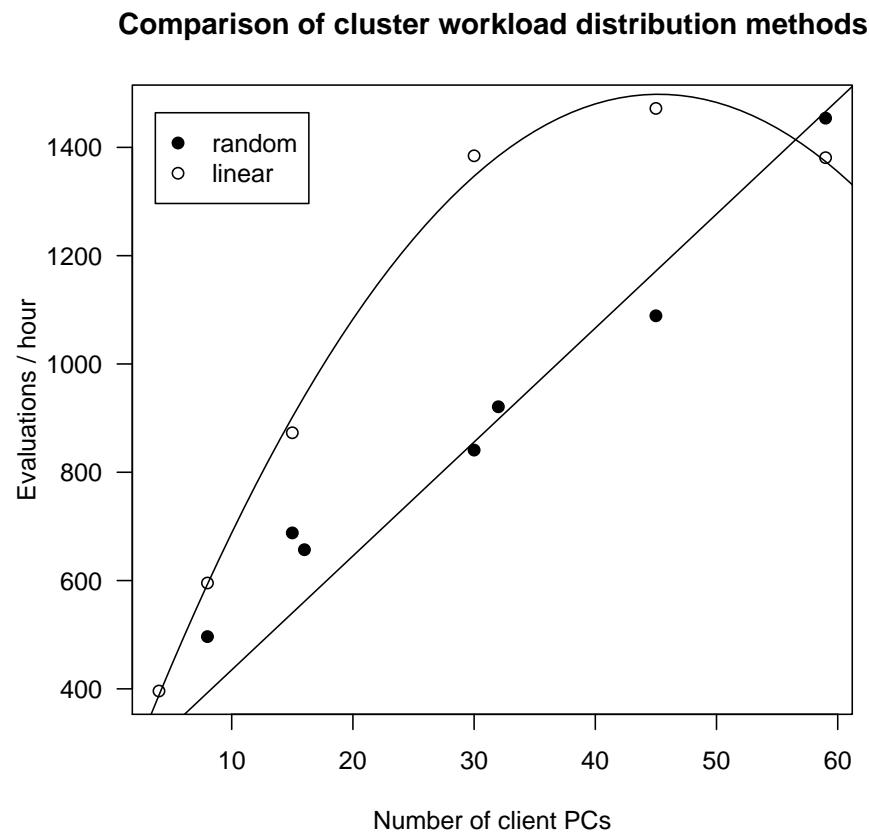


Figure 9.2: *Performance of fitness evaluations on cluster.*

9.3 Summary

This chapter has reported on testing of the software. The response of a neuron to a particular pattern of input behaviour was examined to ensure that the neuron models work and that the quantised models were not completely useless. Each of the implemented neuron models has been tested, with varying levels of arithmetic precision for calculating outputs and internal state. The input stimulus was varied in strength, and switched from being initially inhibitory to excitatory.

The base of reference for a neuron implementation is to use floating-point arithmetic — this is the standard for research, and the type of arithmetic used by the majority of previously published literature.

The first aim of this testing was to verify that the software models were coded correctly, firstly by not crashing, and secondly by producing some output that responded in an expected way to the varying input signal. This was achieved; all of the models appeared to be bug free, and there were obvious changes in output signal given corresponding prior changes in input signal.

The second aim was to see whether reducing the precision from floating-point would dramatically alter the neuron’s behaviour. This test was invalid for the logical neuron, because there is no reason to expect the values within near cells of a randomly initialised lookup table to have any similarities. For all the other models, it was observed that decreasing the precision from floating-point to 6-bit did preserve the overall neuron behaviour and output signal waveform quite well.

The sine “neuron” preserved the shape and period of the output waveform particularly well, since it is just an oscillating function. If behaviour were driven by oscillations rather than by the “shape” of the wave it is entirely possible that a complex sine wave function could be replaced by a simple binary oscillator in some applications. For other models, we see that reduction to a precision of 6-bit preserves the neuron behaviour well, generating very similar internal state and output values. Spiking behaviour is almost identical, with spikes being generated at the same time as with floating-point precision, although the SRM model did display some different spiking behaviour, with one more spike being generated with 6-bit precision.

For most models, 4-bit and 5-bit precision also offered good approximations of the floating-point behaviour, but below that the approximation was poor.

This chapter has shown that reducing the arithmetic precision of a neuron model from floating-point to, say, 6-bit, can result in a model that substantially approximates

the behaviour of the floating-point model but with far fewer states and lower computational requirements. A 6-bit model has 64 states for each variable, whilst a floating-point model has 2^{64} states for each variable, so this is a huge saving in computational resources. Integer or quantised arithmetic units are also simpler to implement.

The amount of verification carried out in this chapter is limited. The models were only tested with a single input pattern set. We have not shown that, for all possible input patterns, or all possible neuron configurations, the approximating behaviour will be preserved. The nature of the neuron input function means that at each time step the neuron is presented with a single value that represents the weighted sum of the current inputs. Using a single input neuron as done here is perfectly valid — this single input is capable of generating the same input patterns *post input function* as multiple neurons with varying weights. This means the testing done here will extend to some larger set of possible input configurations and input patterns. However, it is still apparent that this testing is limited. It is impossible to test all input patterns and all possible neuron configurations. What has been done here is to show that reducing the arithmetic precision massively, from 2^{64} states to 2^6 states, does not completely disrupt the behaviour of a neuron model.

The performance of the distributed genetic algorithm was quantified. With around 55 hosts, 1400 fitness evaluations could be carried out per hour. This is obviously a very rough figure, as the exact number will vary depending on the complexity of the physics simulations being carried out, and on other factors (e.g. availability of computational resources on the shared PC cluster, performance of the database server, etc.). Nevertheless, it was a useful baseline estimate that could be used to judge the computational feasibility of different proposed experimental setups.

Two different algorithms for scheduling fitness evaluations across the cluster were implemented and compared. One algorithm assigned evaluation datasets to clients randomly and made no attempt to avoid repetition of workload. The other algorithm attempted to assign an evaluation dataset only once to avoid repetition of work. It was found, contrary to expectations, that linear assignment performed better when 55 hosts or less were used for client evaluations, but that random assignment scaled better above that. Investigation of the cause established that certain classes of process were not being killed as should have happened when the genetic algorithm client software was given (supposedly) exclusive access to a PC. The computationally intensive processes of other users were occasionally left running in the background. Although this occurred only on a small minority of the PCs, the effect was to delay completion of

the fitness evaluation of a full population, since by design no other host was assigned that particular dataset. In contrast, the “random” scheduling algorithm avoided this problem by making no attempt to avoid duplicating individual datasets; a particularly slow PC would not hold up all the rest, since a faster PC would inevitably be assigned a duplicate of the dataset and complete it earlier.

Chapter 10

Pole balancing experiments

This chapter will describe a set of experiments designed to evolve and quantify the performance of neural networks to control a fixed “cart and pole” morphology carrying out the traditional AI pole balancing control task. The aim of these experiments was to establish whether or not quantised neuron models could be used for this task, and if so, how the performance of quantised models compared to that of floating-point models. The robot morphology consisted of a cart with freedom of linear movement along the x -axis joined to a pole with freedom of angular rotation about the hinge joint. The morphology was fixed and did not need to be created or optimised by evolution.

The previous chapters have described how various control networks operate, and how genetic algorithms can be used to evolve these networks. When connected to a fixed robot morphology within a 3D physics simulator, a fitness task can be devised which evaluates the network’s ability to perform some robot control task. This fitness function can then be used by a genetic algorithm to drive the evolutionary process. This chapter will introduce the first set of experiments that are designed to investigate the qualitative differences between the control abilities of floating-point and quantised neural networks. A linear quadratic regulator (LQR) controller was also implemented as it is a typical engineering solution to the pole balancing problem, and provides a baseline of comparison for the performance of evolved controllers.

10.1 Introduction

The pole balancing task (also known as the “inverted pendulum problem”) is a standard problem in the field of AI control systems [49]. The problem layout is quite simple (see figure 10.1):

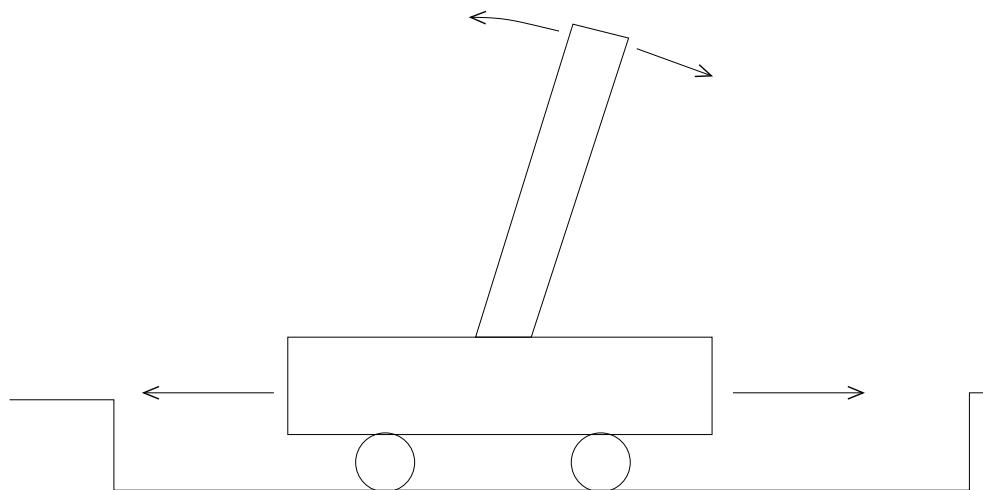


Figure 10.1: *The pole balancing task is a typical AI control problem. A cart and pole are joined by a hinge. The cart can move left and right, but is prohibited from travelling too far. The force of gravity pulls the pole down. The controller must apply a continuous sequence of forces in order to balance the pole and prevent it from falling.*

A cart is allowed to move backwards and forwards along a 1D line. The cart is attached to a pole via a hinge joint on the cart, thus allowing the pole to rotate about the cart. Both the cart and the pole possess mass, and a force due to gravity acts downwards. The angle between the vertical line extending up from the centre of the cart and the pole is initialised to some value close to 0. A neural network controller is attached to the sensors and motor of the cart (figure 10.2). The task of the control system is then to prevent the pole angle from exceeding some threshold by applying a series of negative and positive corrective forces to the cart in order to return the pole to an upright vertical position.

In order to prevent the controller from achieving balance by merely accelerating the cart to a high velocity, stop limits are placed at either side of the cart along the 1D line.

The inputs to the controller can be varied; the following four inputs are typical and were used in the experiments here:

- θ pole angle
- $\dot{\theta}$ pole angular velocity
- x cart displacement along the 1D line
- \dot{x} linear velocity of the cart

Allowing the control network to use different inputs affects the difficulty of solving

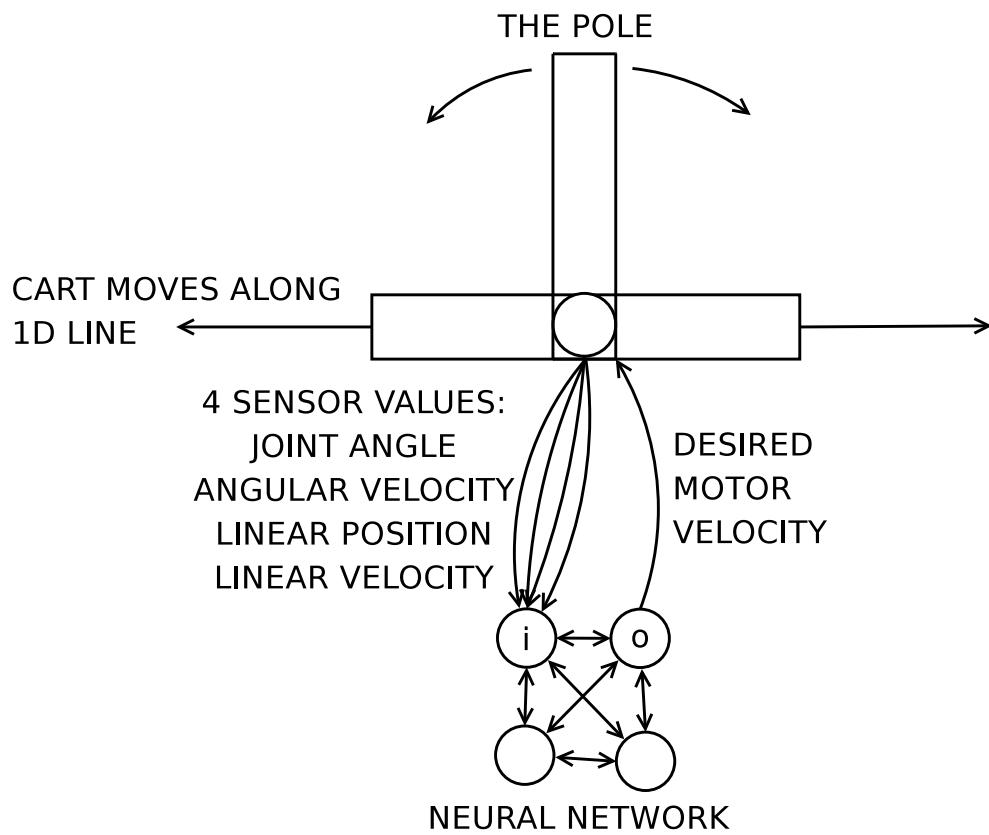


Figure 10.2: The cart is connected to a neural network. Sensors from the cart (e.g. pole angle and velocity, cart location and velocity) can be connected as inputs to the neural network (input neuron labelled “i”). An output neuron (labelled “o”) drives the one degree of freedom motor which pushes the cart backwards or forwards along its line of motion. The neural network size, connectivity and neuron models can be drawn from any of the configurations described in section 8.5 (the configuration given here — internally fully connected — is only one option). The exact parameters of the neural network will be evolved using the genetic algorithm.

the task. For example, a controller that only has the single input θ must attempt to estimate the displacement x in order to avoid the stop limits. Although possible, this makes the task harder. Similarly, knowledge of the angular and linear velocities of the cart and pole allow the controller to apply a more accurate correctional force. Lack of these inputs also makes the task harder.

The task itself can be changed in order to add variation and alter difficulty. The pole can be periodically struck with a random increasing force. The time for which the controller balances the pole can then be used as a performance metric. The pole can be attached to a second pole with a hinge. The task is now to balance both poles, which is substantially harder. Another variation adds a desired x cart displacement which the controller attempts to move towards whilst keeping the pole balanced. A similar task is to keep the pole pointing towards a randomly moving point.

With each of these variations there are several parameters that can also be varied, such as the masses of the cart and pole, the length of the pole, and the coefficients of friction for both linear and angular movement.

10.2 Task

Discrete network controllers were evolved to solve the pole balancing task. Their performance was compared to that of evolved floating-point network controllers, and also to the optimal linear quadratic regulator (LQR) controller. This pole balancing task used a single vertical pole, with stop limits placed to either side of the cart. The pole was periodically hit with a randomly generated force acting on its centre of mass; the maximum amplitude of the force increased linearly over time. Performance was evaluated by recording the number of seconds that the pole remained above $\pm\frac{\pi}{2}$. The simulation would end either when the pole fell, or after 30 simulated seconds if the pole did not fall.

10.3 Task analysis

Several people have evolved continuous networks to solve the pole balancing problem. In 1991 Wieland evolved neural networks to successfully balance not only poles, but also double poles and jointed poles [479]. In 1997 Pasemann solved the pole balancing problem with evolved neural networks using a biologically based co-evolution algorithm to evolve both the number of neurons and the connectivity in a continuous

sigmoid network [325]. In 2000 Pollack used staged evolution to first evolve a pole balancing neural network. A hard spring was then introduced in the centre of the pole, and slowly relaxed as evolution proceeded. This evolved neural networks which could solve the double pole balancing problem [340]. In 2004 Stanley used the NEAT algorithm to evolve solutions to the double pole balancing problem [409]. In 2005 Gomez extended the pole balancing problem to three dimensions, and successfully evolved neural network controllers [163].

10.4 LQR controller design

The linear quadratic regulator (LQR) was constructed following the method in [292]. The equations of motion for the pole balancing task are [49] :

$$\ddot{\theta}_t = \frac{g \sin \theta_t + \cos \theta_t \left[\frac{-F_t - m_p l \dot{\theta}_t^2 \sin \theta_t}{m_c + m_p} \right]}{l \left[\frac{4}{3} - \frac{m_p \cos^2 \theta_t}{m_c + m_p} \right]}$$

$$\ddot{x}_t = \frac{F_t + m_p l \left[\dot{\theta}_t^2 \sin \theta_t - \ddot{\theta}_t \cos \theta_t \right]}{m_c + m_p}$$

θ_t $\dot{\theta}_t$ $\ddot{\theta}_t$ pole angle, angular velocity, acceleration

x \dot{x} \ddot{x} pole position, linear velocity, acceleration

g gravitational acceleration ($9.8ms^{-2}$)

m_c m_p mass of cart and pole

l length of pole

t time

F_t horizontal force applied to cart at time t

These equations can be linearised about 0° and represented in state space format (as in [292], but disregarding pole inertia and cart friction):

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -m_p l^2 & m_p^2 g l^2 & 0 \\ 0 & \frac{m_c + m_p + m_c m_p l^2}{m_c + m_p + m_c m_p l^2} & \frac{m_c + m_p + m_c m_p l^2}{m_c + m_p + m_c m_p l^2} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -m_p l & \frac{m_p g l (m_c + m_p)}{m_c + m_p + m_c m_p l^2} & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ m_p l^2 \\ \frac{m_c + m_p + m_c m_p l^2}{m_c + m_p + m_c m_p l^2} \\ 0 \\ m_p l \end{bmatrix} F_t$$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} F_t$$

This was implemented as a continuous state space system in the GNU Octave programming language, and converted to a discrete system using zero-order hold sampling. The sample frequency was $\frac{1}{50}$ second, which corresponds to the frequency of physics simulation with a step size of 20ms. A discrete LQR controller was then created using a state cost matrix that heavily weights the cart position (to keep it in the centre) and puts a lesser weight on the pole angle (to keep it up). This returned a control matrix which was translated into the equivalent Python matrix operations and incorporated into the physics simulator.

10.5 Experimental design

The experiments aimed to test whether there was any observable difference between the different models of neuron when used in the pole balance control task, whether there was any difference between the floating-point neuron models and quantised models, and how other factors, such as the neural network topology, timing model, and genetic algorithm parameters might influence the results. The null hypothesis for each factor (or interaction of factors) is that all levels of the factor are equal, so mean values for the levels will all be the same. The alternative hypothesis is that the levels of a particular factor are not equal — that one or more have mean values that are different from the rest.

It was decided to carry out a “factorial experiment design”, in which all of the factors are varied simultaneously. This has several advantages. The number of experiment replicates is minimised, which is desirable here since evolution is computationally intensive and there were constraints on the computational resources available. Varying

multiple factors simultaneously enables potential identification of potential interactions between different factors. Another advantage of a factorial design is that, from the perspective of a single factor, the other factors are effectively randomised. This removes bias of the results due to other factors, and allows the effect of individual factors to be more accurately estimated. In a “full factorial design”, all possible combinations of all levels of all factors will be sampled. This was not possible here since running a number of simulation replicates equal to the total number of possible combinations would be intractable. A “fractional factorial design” would instead be used, where the space of possible replicates from a “full factorial design” design is sampled to create a smaller design with fewer replicates.

A fractional factorial experimental design was created using the *AlgDesign* package for R [43,477]. A full factorial design was first created, sampling every point in the factor space. This design was then fractioned using the Federov optimiser to reduce the number of replicates whilst preserving optimal sampling. The total number of trials was chosen to sample two-factor interactions at least 10 times. Once finalised, the experimental design was dumped to a file, and then processed by a script to generate a sequence of calls to create evolutionary populations with the necessary parameters. Each run was then despatched to the computational cluster for processing.

Some parts of the search space were removed; the topology does not affect networks using the sine neuron model. Networks with four neurons and a 2D geometry with size four neighbourhood are equivalent to fully connected, and so were removed. Logical networks with more than 8 states, or with full connectivity between more than four neurons, were removed as the size of the lookup table required for each neuron increases exponentially (i.e. $4^8 = 64,000$ entries for an 8 input neuron with four quanta states), making large numbers computationally intractable. To counter this reduction, the valid points in the “logical” model search space were replicated twice.

The final fractional factorial design consisted of 533 replicates. The table shows the factor levels evaluated in the design:

FACTORS	NO. LEVELS	LEVELS
model	7	sigmoid, logical, beer, if, ekeberg, sine, srm, taga
quanta	7	2, 4, 8, 16, 32, 64, fp
num neurons	3	4, 9, 16
sync	2	sync, async
mutate type	2	gauss, uniform
mutate prob	2	0.01, 0.05
gen/pop size	2	50, 100
topology	6	full, 1d-r1, 2d-r1, nk1, nk2, nk3

The factors are:

model The neuron model being used.

quanta The type of neuron arithmetic modelled. Either the number of quanta states for a quantised model, or “fp” for a floating-point model.

num neurons The number of neurons in each network.

sync Timing model of the network. With synchronous timing the output signal value of every neuron is updated simultaneously, and changes do not become visible until all neurons have been updated. With asynchronous timing the output signal value of each neuron is updated one at a time, and changes become visible immediately.

mutate type The type of mutation performed on parameters, either Gaussian or replacement from some range with uniform probability.

mutate prob The probability of mutation occurring for any given parameter.

gen/pop size A combined metric of population size and number of generations to run the evolutionary algorithm for. This parameter effectively measures computational power, as it directly relates to the number of fitness evaluations.

topology How the neurons in the networks are connected:

full Fully connected.

1d-r1 One dimensional geometry with a Moore neighbourhood of size 3.

2d-r1 Two dimensional geometry with a Moore neighbourhood of size 9.

nk1 Random network with $k = 1$ (k being the number of inputs to each neuron from other neurons (connections from sensors and to motors are not fixed, so this is not a strict k input topology)).

nk2 Random network with $k = 2$.

nk3 Random network with $k = 3$.

For more information on these factors and their levels, see chapter 8.

Note that the generation size and population size were considered together as a single factor. The reasoning for this was that both are actually directly related to the larger concept of “computational capacity”. The number of fitness evaluations carried out is a product of both the generation size and population size; hence both are ultimately just a factor of the real metric that we are interested in. There was also the major consideration of computational resources available for these experiments — producing statistically valid experimental runs that vary the population size and number of generations independently across multiple levels would have required considerably more computational capacity than was available.

Choosing the number of generations and population size is a bit of a black art; researchers generally use numbers that are as large as computational constraints allow. Some typical examples of (population size, number of generations) from evolvable creature research include Bongard (200, 200) [35], Chaumont (300, 100) [60, 61], Hornby (100, 100-500) [200, 204, 205], and Sims (300, 100) [397, 398]. In the case of this design, the population size and number of generations were varied between the two levels (50,50) and (100,100). These numbers were on the limit of what was possible given that the experiments were to last no more than a couple of weeks, and would provide an answer to the question as to whether going higher than (50,50) would result in measurably better controllers.

The neural network configuration was varied between different geometric layouts and connectivity neighbourhoods — combinations of fully connected, one-dimensional and two-dimensional with different size neighbourhoods, and random networks with 1, 2, and 3 inputs per node. See section 8.6 for an explanation of the difference between the neural network geometry and connectivity neighbourhood. Each topology had different parameters subject to evolution and optimisation by the genetic algorithm, see section 8.8 for details.

The genetic algorithm was fixed to use a generational algorithm as opposed to steady-state. The only factor varied for the LQR controller was the number of quanta

states. For the LQR controller, ten replicates were run for each quanta level.

10.6 Reproducibility

The complete experimental setup described here consists of thousands of lines of source code. It is not possible to give a rigorous definition in the English language or in pseudo-code that would allow this to be reproduced exactly — not only would such a description be lengthy, but the nature of software simulation means that deviations in even small detail from the original experimental setup may cause some changes in the observed results. To aid increased openness and experimental reproducibility, the exact source code and scripts used to run this experiment will be published along with this thesis. Section 1.3 contains some more commentary on the reproducibility of computer simulations and the importance of this to the scientific process.

10.7 Results

At the end of each experimental replicate the “score” (the number of seconds that the pole had remained balanced) of the highest performing individual was recorded. Only the highest scoring individual from the final generation of each replicate was analysed — the particular distribution of the other individuals in the final generation was not considered important, as we are interested in obtaining the best controllers — it may be the case that some evolutionary algorithm leads to populations having a small number of high fitness individuals and a large number of low fitness individuals, but the fact that the population is mostly composed of low fitness individuals is irrelevant, since for practical purposes (like actually selecting a controller for a physical robot) the selected controller will be the one that ranks highest on the tasks that it must carry out. This is not to suggest that the other controllers in the population are useless, merely that, under the fitness criteria being considered, they were not as good as the highest scoring controller.

10.7.1 ANOVA modelling

The resulting dataset was loaded into the *R* statistics software and processed. An “Analysis of Variance” (ANOVA) model was created. The ANOVA model assumes that replicates are truly independent. In the pole balancing experiment carried out here,

each replicate was digitally simulated using a new simulation environment, and hence the replicates were truly independent. The ANOVA model also assumes that scores around the mean values of different sub-populations in the data (the *residual error*) are normally distributed and of equal variance. However, ANOVA is robust to minor variations in its assumptions of normality and constancy of variance, and it is usually recommended to merely plot the data to confirm that the data has a distribution which is close to normal, rather than to carry out one of the stricter formal tests of normality, which would classify many real world data with trivial departures from normality as being non-normal even though the robustness of ANOVA means it would be suitable for modelling such data [8, 77]. It is also recommended to plot ANOVA fitted mean values against variance and to judge visually whether there are major violations of the assumption of constant variance. If there are any deviations from these assumptions, it is sometimes possible to transform the data so that it better satisfies the assumptions.

The residuals of the ANOVA model were plotted as a histogram (to check distribution shape), against fitted values (to check whether variance changes with mean fitted values), and as a Normal Quantile-Quantile (Q-Q) plot (to check normality) (figure 10.3). The initial plots showed that there was some small non-normality, and some heteroscedasticity, where the variance increased with the mean of fitted values. The Normal Q-Q plot appeared linear in the centre of the data, but departed from the fitted line at both high and low values. The histogram suggests that this distinctive shape is caused by “long tails” in the distribution (there are significant values at the extremes of the x -axis, the values below -6 stand out visually).

The dataset was transformed using a log function to attempt to correct the small deviations from non-normality and heteroscedasticity. An ANOVA model was generated from the transformed data. The residual distribution showed better fit with normality, more uniform variance, and the Normal Q-Q plot was more linear (figure 10.3). Although there are visible outliers, the vast majority of values are linear — of the 533 replicates plotted there are only a few visible outliers, meaning that most are part of the superimposed dots forming a thick line in the centre of the plot. In the end, the only significant effect of the transform was to lower the probability of accepting the *timing* factor as being statistically significant. Before the transform, *timing* was highly significant with $p = 0.0004$, afterwards the p value was 0.044. A Kruskal-Wallis rank sum test, which is resistant to non-normality, gave a p value of 0.023, a Welch Two Sample t -test, which can be used to compare samples with heterogeneous variances, gave a p value of 0.045, and a Wilcoxon rank sum test with continuity correction gave

a *p* value of 0.023. Thus it was decided that *timing* should be accepted as a significant factor, as the *p* value was below the 5% level by all measures.

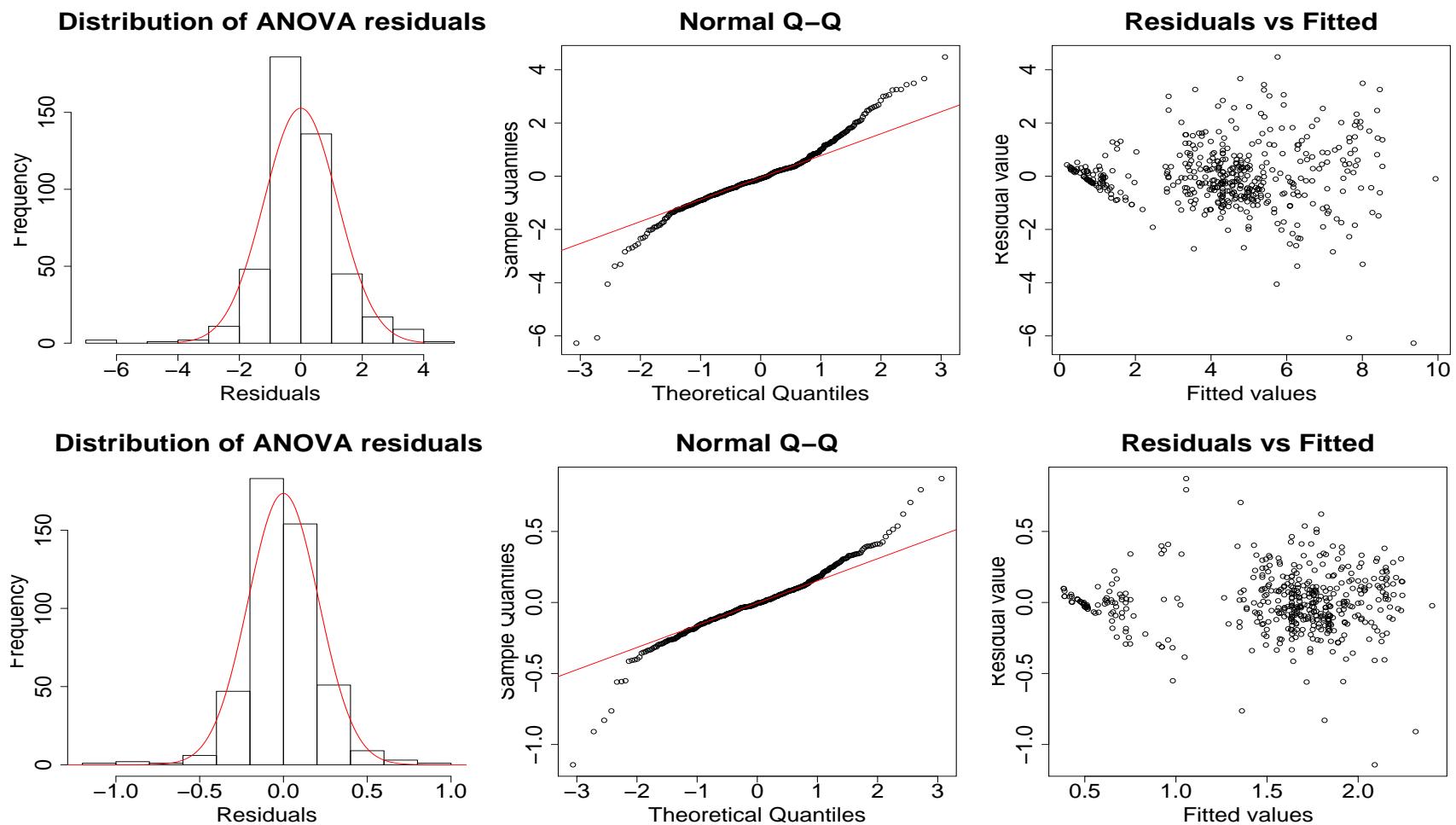


Figure 10.3: ANOVA residuals initially showed some non-normality and heteroscedasticity (top) which was improved with a log transform (bottom), producing smoother tails, a more linear Normal Q-Q plot, and more slightly widely varying residuals for low fitted values.

The non-normality and heteroscedasticity was contributed to greatly by small score values, produced by controllers that failed to react to the changing pole and simply allowed it to fall. The score of poles that immediately fell had less variance than those that participated in a longer simulation. If scores less than 2.5 seconds were removed from the dataset, the resulting data would have been much more normal. However, this would have meant removing a significant number of replicates (120), which would have negatively affected the statistical analysis.

One of the aims of a multi-factorial experimental design is to randomise all levels of factors that may be significant. It then becomes possible to compare the different levels of a single factor to determine whether they differ significantly. The randomisation of the other factors with respect to the factor under study means any biasing of the results due to those other factors will be minimised. A linear regression model can be used to determine whether changes in the levels of a single factor produce significantly changed results. The linear regression model in *R* works by calculating a coefficient for each term in the linear equation which corresponds to the first level of a factor (the *intercept*), and then using Student's *t*-test to compare the population at this level to those of each of the other levels in turn [77]. The model will show which levels are statistically different from the base *intercept* for a particular factor, however, it does not immediately show differences between other levels, as it is up to the human operator to examine further by generating a more complete regression model and comparing it to the original.

10.7.2 ANOVA results

The null hypothesis for each factor was that there was no difference between the mean scores of controllers where that factor was set to different levels. The alternative hypothesis was that there was a difference. The significance level used in this work was 5%. The ANOVA model showed that all of the below factors were statistically significant at the 5% level. The *p* value was very low for all factors apart from *timing* — they would also have been significant at the 1% level. As already noted, the *p* value for the transformed *timing* dataset was 4.4%. The significant factors and corresponding *p* values were:

Factor	Significance (p)
model	$\leq 2.2\text{e-}16$
q	$2.2\text{e-}16$
genpop	$3.696\text{e-}11$
timing	0.044174
model:genpop	$2.635\text{e-}08$
model:timing	$1.667\text{e-}06$
model:q	$1.619\text{e-}15$

(the colon character denotes an interaction between two factors)

All of the other factors and possible combinations of factors were found to be not significant at the 5% level.

10.7.3 About “Least Significance Difference” plots

Visual comparisons of the different levels of a single factor can be carried out by plotting the data. This is traditionally done with either a box and whisker plot, or with a bar plot with error bars. Box and whisker plots show the quartiles and extremes of the data, but do not visualise any statistically significant difference between the means. Bar plots can be used to plot the mean values, but do not give as much information about the overall shape of each distribution. Error bars can be added to each bar in the bar plot, and the length of the error bars can be used to indicate information to help compare means visually. The question arises as to which data when used as the error bar length best conveys statistically significant differences (or not) between the mean values of different levels? Crawley notes that using ± 1 standard error as the length of the error bars means that overlapping bars imply that the means are not statistically different at the 5% level, but the reverse is not true — not overlapping does not imply statistical difference [77, page 169]. Using 95% confidence intervals indicates that the means are significantly different when the bars do not overlap, but the reverse is not true — overlapping bars does not imply no statistical difference. Crawley suggests using *Least Significant Difference* error bars [77].

With *Least Significant Difference* error bars, non-overlapping bars indicate a significant difference at the 5% level, and overlapping bars indicate no significant difference. The *Least Significant Difference* is calculated as $LSD = qt(0.975, df) \times s.e.\text{difference} \approx 2s.e.\text{difference}$ (where qt is the quantile function of Student’s t -distribution, df is the degrees of freedom, and $s.e.\text{difference}$ is the standard error of the difference between two

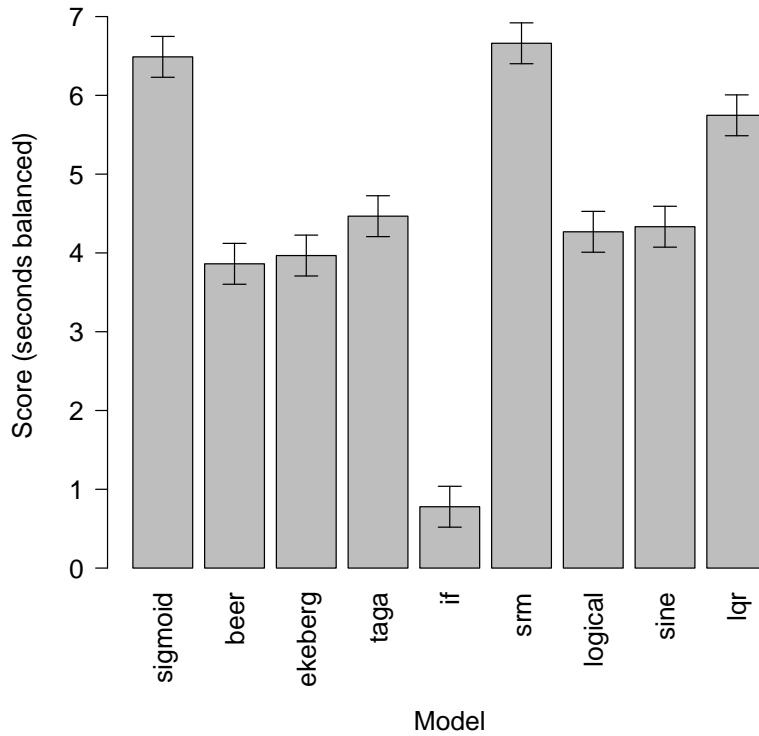
means). The *Least Significant Difference* is then plotted for each mean as $\pm \frac{LSD}{2}$, so that the point where the top and bottom (or vice versa) of the error bars of two means are level represents the position where the means are the *Least Significant Difference* apart — if the means were closer, the error bars would overlap, indicating no difference between the means at the 5% level, and conversely, if the means were further apart, the error bars would not overlap, indicating a statistically significant difference between the means at the 5% level.

10.7.4 “Least Significant Difference” plots

The following pages show *Least Significant Difference* graphs comparing the different levels of each significant factor. The values being compared are the mean scores (number of seconds pole was balanced) averaged across all results where the factor was set to that particular level (this is the standard comparison for a *factorial experiment* [43]). The *Least Significant Difference* error bars denote whether or not the difference between two means is significant at 5% — overlapping error bars imply no significant difference, non-overlapping error bars imply a significant difference.

The *Least Significant Difference* plot is a quick way of visualising and comparing all of the levels of a factor. The metric being compared is the “score” — the number of seconds that the pole was balanced before falling. In all of these comparisons, the null hypothesis is that there is no difference in the sample means of the “score” for experiments with different levels of a particular factor. The alternative hypothesis is that there is a difference between the means of some of the levels of the factor. The significance level used for comparison is 5%. This level is commonly used in genetic algorithms research. The high computational requirements of carrying out many simulation replicates and the inherent variance in the task mean that it is not feasible to carry out the required number of experiment replicates for a 1% significance test. Rather than merely reject the null hypothesis, it is better to state the probability of the observed data occurring if the means were actually the same [77, p.78]. The probability value will therefore be stated for comparisons of levels where the presence of significance may be hard to establish visually, i.e. when the mean values differ with a p value near to 0.05.

10.7.5 Factor: Model

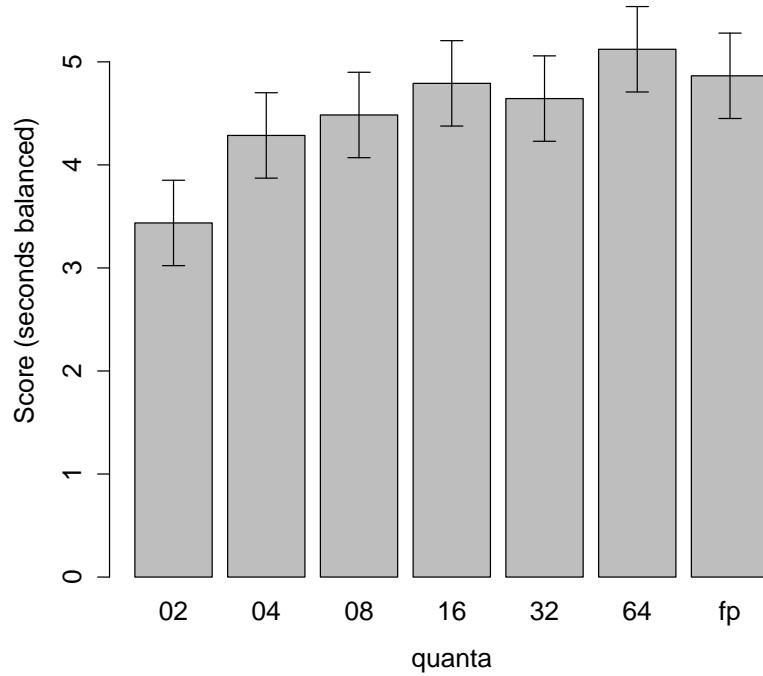


There are statistically significant deviations at the 5% level between the different sets of integrate-and-fire, (beer, ekeberg, taga, logical, sine), lqr, and (sigmoid, srm). The SRM and sigmoid models perform the best, and both outperform the LQR controller. There was no significant difference between the performance of the sine neuron controller and any of Beer's CTRNN model, the Ekeberg model, the Taga model or the logical model. This suggests that for this task, pure oscillating behaviour which ignores inputs can be a reasonable strategy for balancing a pole. However, it is also clear that the evolved sigmoid and SRM models and the LQR model outperform the sine neuron, showing that there is an advantage in reacting to the input signal.

The poor performance of the integrate-and-fire neuron indicates that in its present configuration it is not able to adequately control the pole. This could be down to the neuron model itself failing to produce any useful internal behaviour, or the space of useful behaviour being so small as to be missed by the genetic algorithm. The idea that the problem may lie in the mapping function that converts output neuron spikes to motor control is contradicted by the good performance of the SRM model, which is also spiking and uses the same function. Having said that, it is still possible that there is some undesirable interaction between the integrate-and-fire model and the spike to

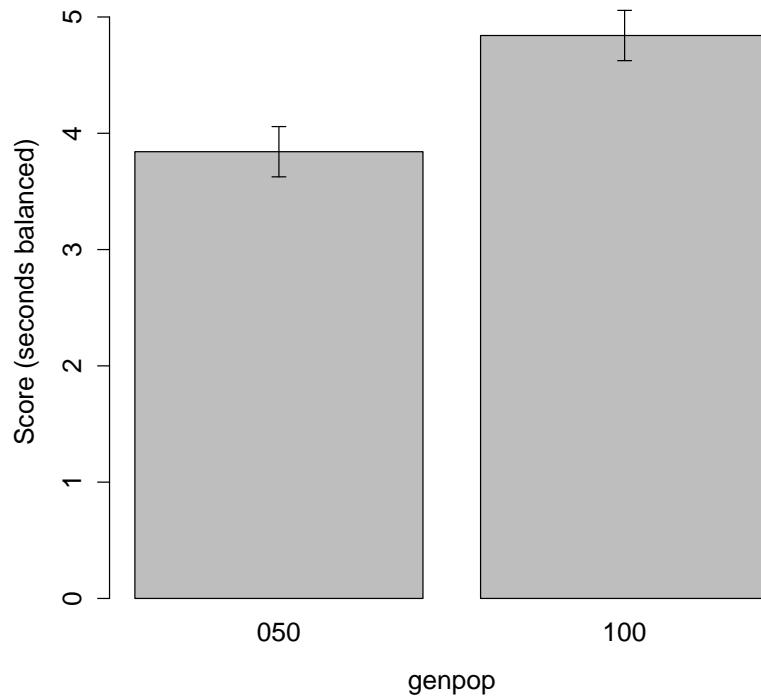
motor control mapping function, and there may be some alternative function which would perform better. It is standard in evolutionary robotics for the functions that handle the mapping of sensory input and motor output between the physics simulator and the neural network to be fixed. One obvious avenue for future research would be to co-evolve such functions along with the morphology and control.

10.7.6 Factor: Quanta



Two quanta (binary) controllers are significantly worse at the 5% level than most of the rest ($p = 0.02$). There is no statistically significant difference between the others. This is an interesting result, as it shows that, in general, the performance of evolved controllers does not decrease when a quantised model is introduced, and even when that model is reduced to using only 4 quanta states. This suggests that, for the pole balancing task, floating-point arithmetic is not a necessary prerequisite for creating high performance evolved controllers, and that quantised neural controllers, with their lower complexity and power requirements, should be considered.

10.7.7 Factor: Number of generations and population size



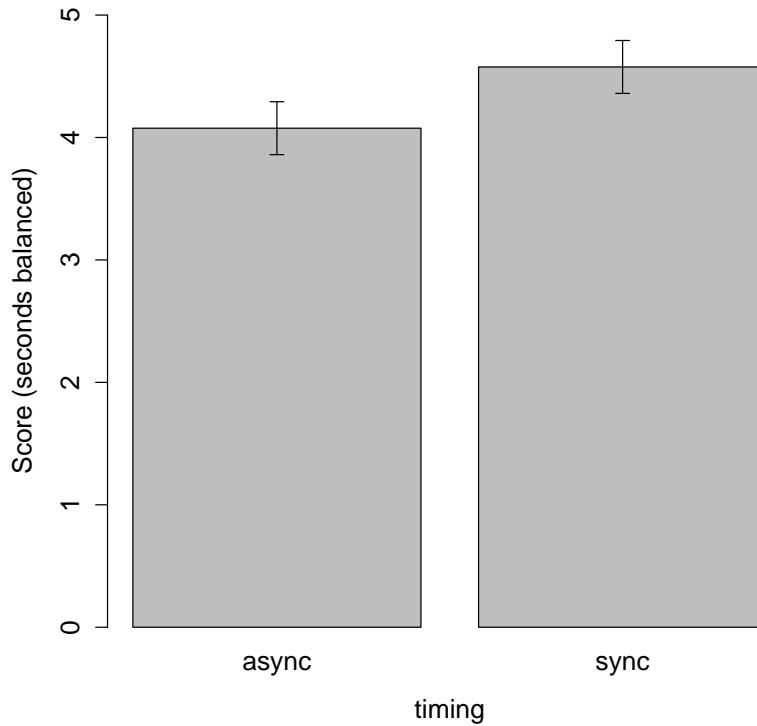
Using a population size and generations of 100 is significantly better than one of 50 at the 5% level. Population size and generations are representative factors of the underlying concept of computational capacity, as they directly relate to the number of fitness evaluations that have to be carried out by the genetic algorithm.

In evolutionary robotics research the population size and generations are usually chosen based upon the computational capacity available and the estimated time to complete to complete the experiments. Beyond that, little justification is usually given for choosing particular values, and often values are chosen simply because they were used in some previously published work on a similar topic. It was desired here to test whether there was any point in using more computational resources than required by a size 50 population run for 50 generations.

If there were no advantage to be gained by an increase to a size 100 population run for 100 generations (an increase of four times the computational requirements) then we would have establish a rough upper bound on this type of evolution. The results show that this is not the case — increasing the computational capacity did result in evolved controllers with a higher fitness. Since there is no comparison point beyond a size 100 population evolved for 100 generations, we can not say whether or not this

level might represent some upper bound. Going higher than the 100×100 case, or exploring many alternative levels, was not practical for the experiments described here due to the increased computational resources it would require.

10.7.8 Factor: Timing

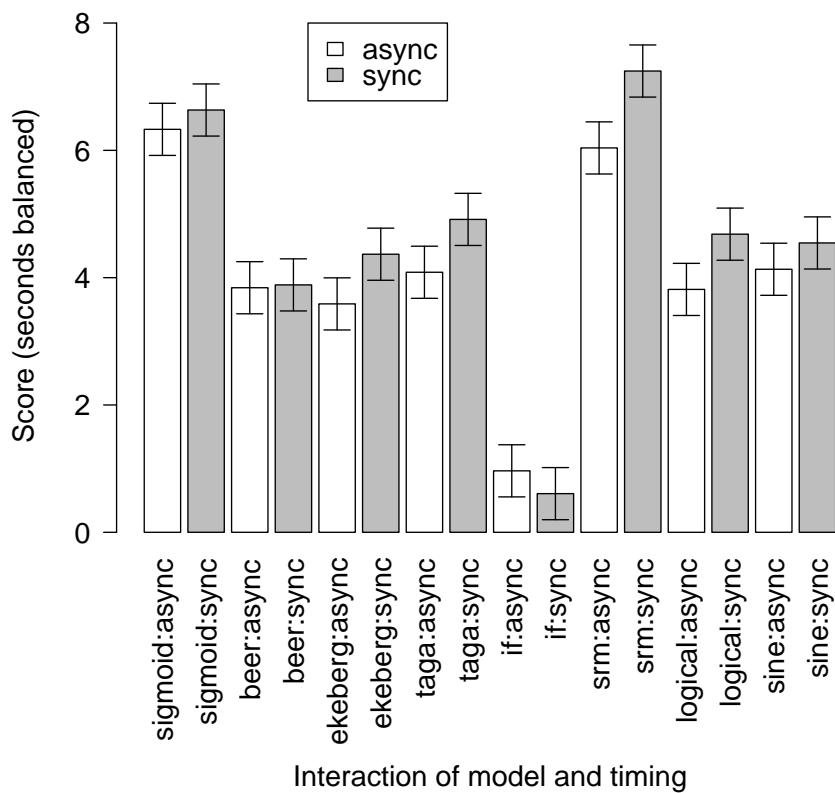


Synchronous controllers are better at the 5% level ($p = 0.16$). This is an interesting result as it shows that, even with such a seemingly simple task, where networks have only four inputs and one output, a synchronous neuron update scheme outperforms an asynchronous one. This strongly suggests that some degree of synchronisation is desirable for neural control networks. In the real world, problems with power consumption and signal distribution mean that a global timing signal is undesirable. It may be better to use a hybrid approach where only localised synchronisation is used. These results show that completely abandoning synchronisation negatively impacts performance on this control task.

More research is required to identify optimal synchronisation schemes. It is thought that biological neural networks combine values and synchronisation into a continuous real-value signal (see section 3.3). The research field of asynchronous circuits has introduced different encoding schemes which combine value and synchronisation

into one or more digital signals, along with explicit synchronisation primitives (see section 4.3. It would be interesting to find out whether these encoding schemes and primitives could be exploited by a genetic algorithm to produce hybrid synchronisation schemes, where the scope of synchronisation is penalised by the genetic algorithm to encourage minimal optimal timing schemes to emerge. It would also be interesting to attempt to co-evolve a timing function along with the rest of the genome rather than using preset schemes.

10.7.9 Factor: Interaction of model and timing

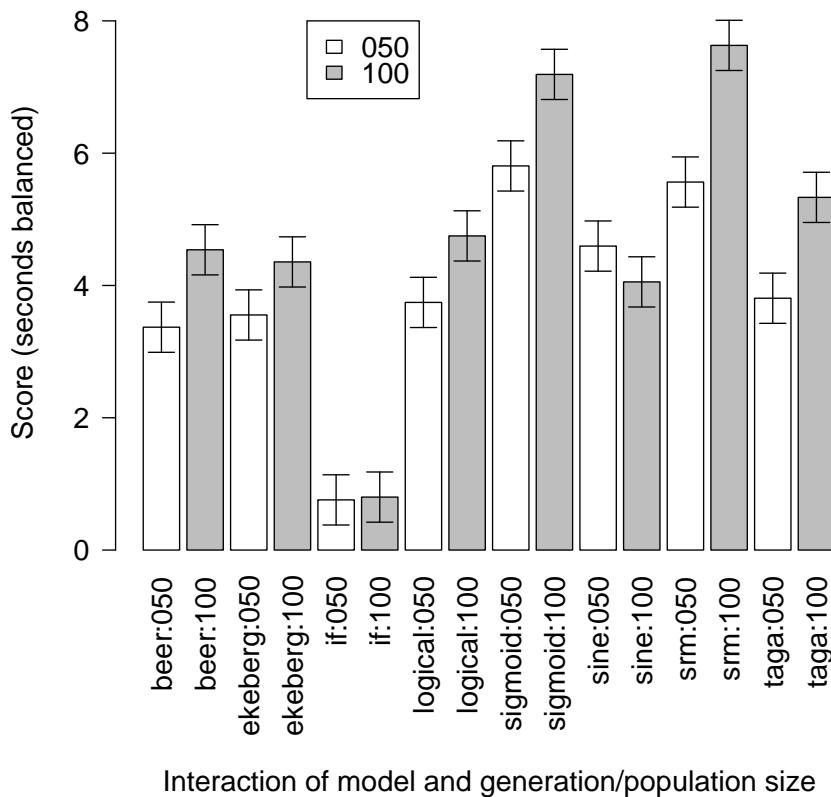


This factor represents the interaction between the neuron model and whether the timing is synchronous or asynchronous. We already know that the *timing* factor is significant, and that on average (across all models) synchronous controllers outperformed asynchronous ones. What this interaction factor shows is whether synchronous controllers outperformed asynchronous controllers for individual neuron models. There is a clear trend that synchronous is better than asynchronous for all models except integrate-and-fire (if), although the difference is only statistically significant at a 5%

level for the Taga ($p = 0.035$) and SRM ($p = 0.004$) models. The difference for Ekeberg is not significant at the 5% level ($p = 0.057$), but it seems likely that more replicates would have established significance. The difference for the logical model is also not significant at 5% ($p = 0.082$).

This result shows that the neuron model and the timing model are not independent, but should be considered together. For some neuron models, the lack of global synchronisation has no effect on this pole balancing control task. It is doubtful that the same result — that synchronisation has no significant observable effect — would be found for all other tasks and possible configurations; it would be certainly be unexpected if synchronisation had no effect even on the scale of millions of neurons.

10.7.10 Factor: Interaction of neuron model and generations / population size



The results have already shown that a combined generation and population size of 100 results in higher scoring controllers than a generation and population size of 50. This factor — the interaction of model and generations/population size — indicates

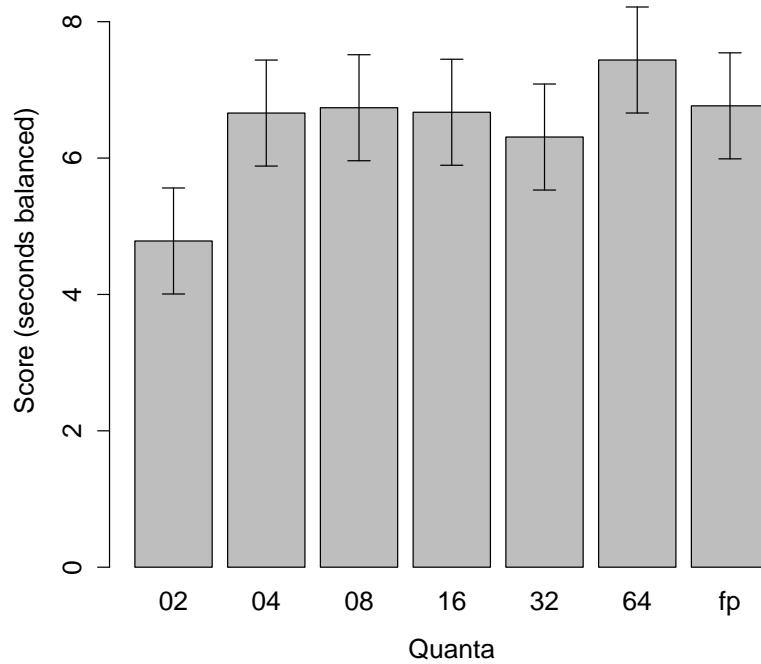
that using a combined size of 100 does not always result in increased fitness over a sizes, and that the neuron model being used has to be taken into account. The results here show that, for most models there is indeed an observable improvement at the 5% level when going from a size of 50 to 100. This would already be accounted for by the existing factor representing the “generation/population size”. However, this interaction factor is significant because for two neuron models — integrate-and-fire and sine — there is no difference between having a size 50 generation/population and a size 100 one.

It appears that integrate-and-fire neurons always perform badly on this task regardless of making extra computational power available to the genetic algorithm. For the sine model, performance is decent, but there is no observable difference when increasing computational effort above the 50/50 level. This is because the parameter space of the sine model is much smaller than the space of the other models, making it easier for the genetic algorithm to locate and optimize a good solution. A population size of 50 evolved for 50 generations is already sufficient to find the best controllers for this task, and no extra effort will result in better performing controllers.

10.7.11 Factor: Interaction of neuron model and quantisation

The performance of each quantised neuron model is individually plotted against the number of quanta states (corresponding to the arithmetic precision of the model). The null hypothesis is that there is no difference between the different quanta levels. The alternative hypothesis is that the means of two or more levels are different. The significance level used for comparison is 5%.

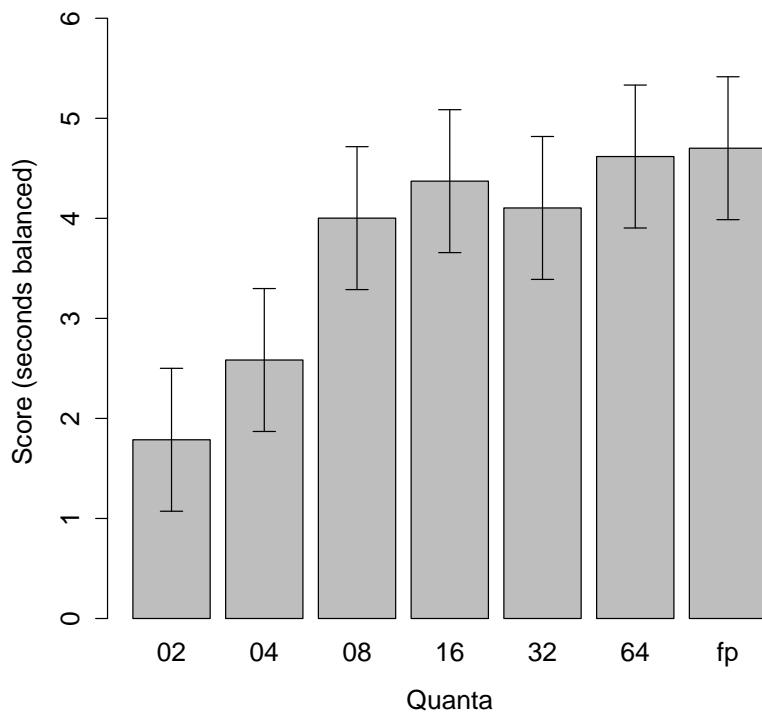
10.7.11.1 Sigmoid model



Two state (binary) controllers are significantly worse than all the rest at the 5% level. There is no significant difference between any of the others, including the floating-point controller. The performance of the sigmoid controllers is high, suggesting that the sigmoid neuron model should be used on this task (in fact, sigmoid and SRM were the two top performing models). The fact that no significant difference was observed in going from floating-point to a quantised 2-bit (4 quanta) model, and the general good performance of this model, means that a low-precision quantised sigmoid neural network should be seriously considered for this control task.

The sigmoid model is widely used in evolutionary robotics research. Implementations always use floating-point arithmetic. These results suggest that the quantised sigmoid model should be considered for other evolved control tasks. Further experimentation is needed to establish whether the success of quantised sigmoid neurons is specific to the pole balancing task, or whether it will generalise to other dynamic control tasks.

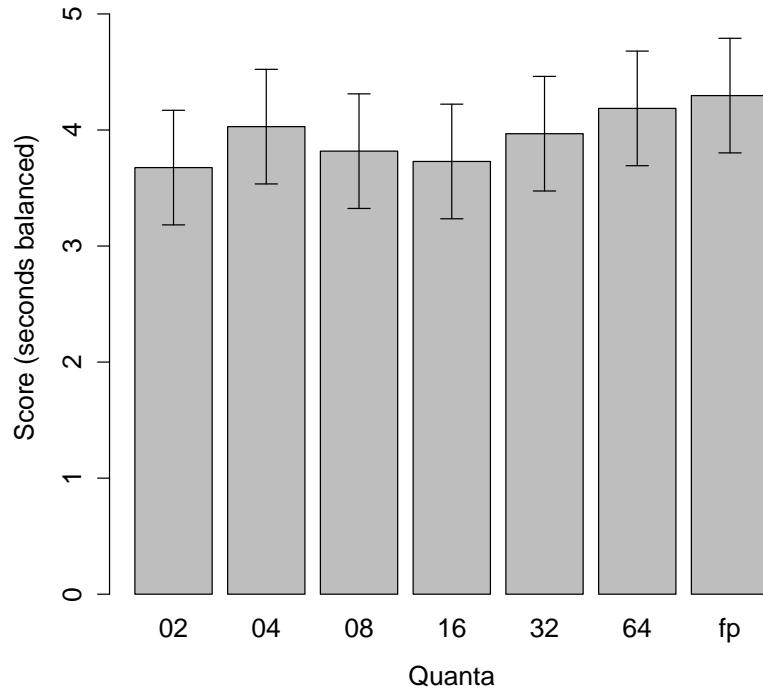
10.7.11.2 Beer's CTRNN model



Two quanta controllers are worse than the all the rest apart from four quanta. The difference between 4 and 8 has a p value of 0.062, and so is not significant at the 5% level. All of the levels with 8 quanta states and above are significantly better than those with 4 quanta states or less.

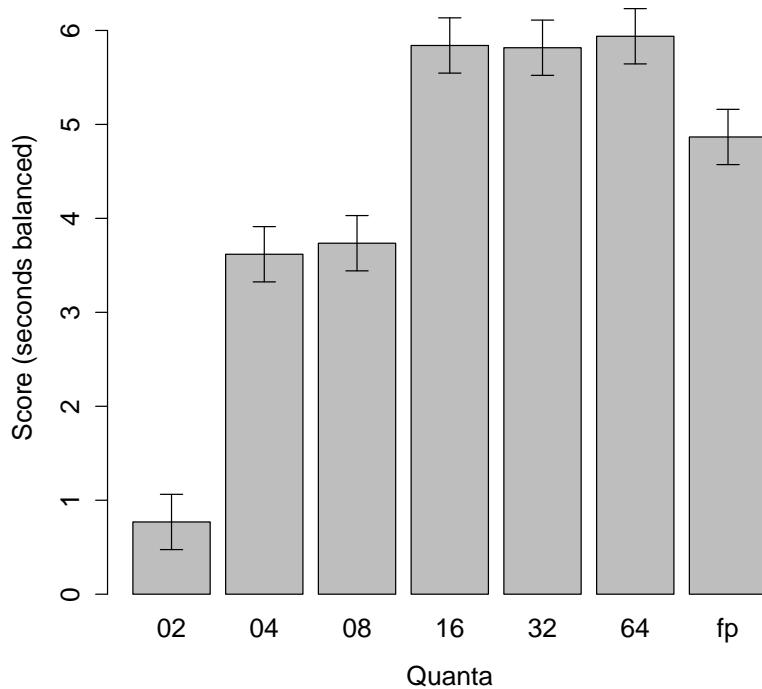
This result shows that for Beer's CTRNN neuron model a floating-point pole balancing controller could potentially be replaced with one using only 3-bit (8 quanta) arithmetic without loss of performance. This would result in significantly lowered complexity and power consumption.

10.7.11.3 Ekeberg's model



There is no statistically significant difference between any of the controllers at the 5% level. This means that for the pole balancing task, an evolved floating-point controller using Ekeberg model neurons could be replaced with a quantised 1-bit (2 quanta) Ekeberg model controller without loss of performance. This is very interesting, as the 1-bit Ekeberg model actually performs reasonably well.

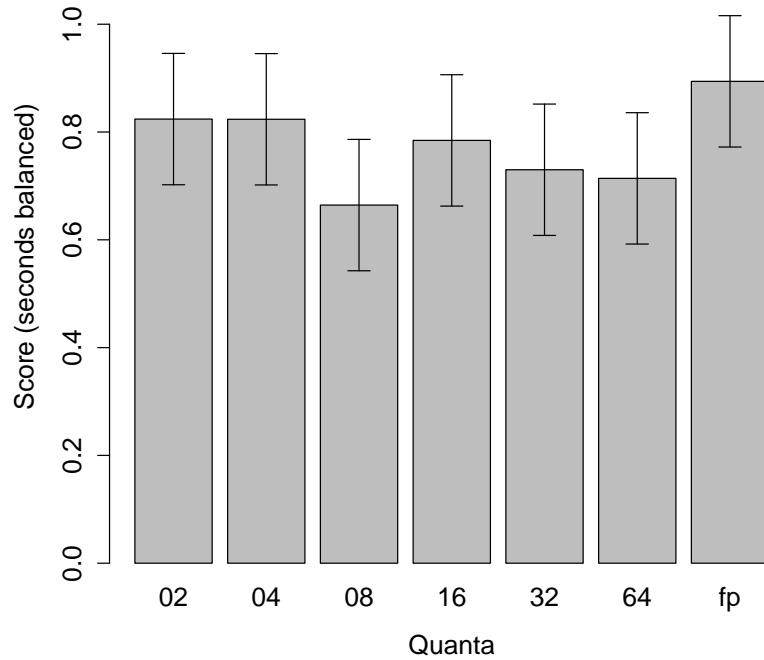
10.7.11.4 Taga's model



There is no significant difference at the 5% level between controllers with quanta levels 16, 32 and 64, although interestingly these all perform better than the floating-point controller. With lower precision, 4 and 8 quanta controllers perform equally well, with the 2 quanta controller performing the worst. It is somewhat unusual that there is an interaction between the neuron model and quantisation that enables three of the quantised controllers to perform better than the floating-point controller. This implies that a floating-point neuron controller carrying out the pole balancing task could be replaced with an evolved quantised controller and performance would actually increase.

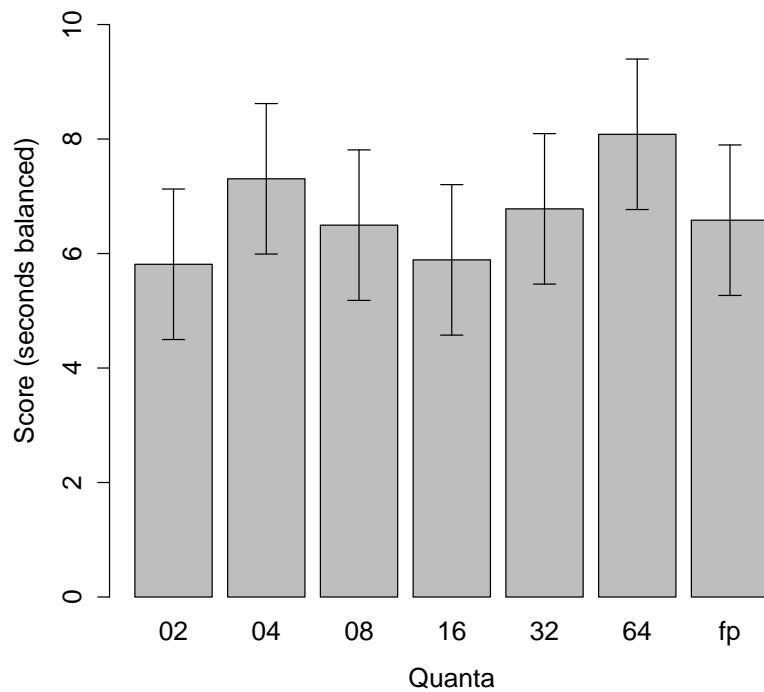
Further research is needed to understand exactly why the floating-point controller performs less well than some of the quantised controllers. There is a significant difference once the number of quanta states drops below 16, with performance degrading substantially. These results indicate that it may be best to use a 4-bit (16 quanta) Taga model, resulting in improved performance over the floating-point model, and lowered complexity and power consumption.

10.7.11.5 Integrate-and-fire model



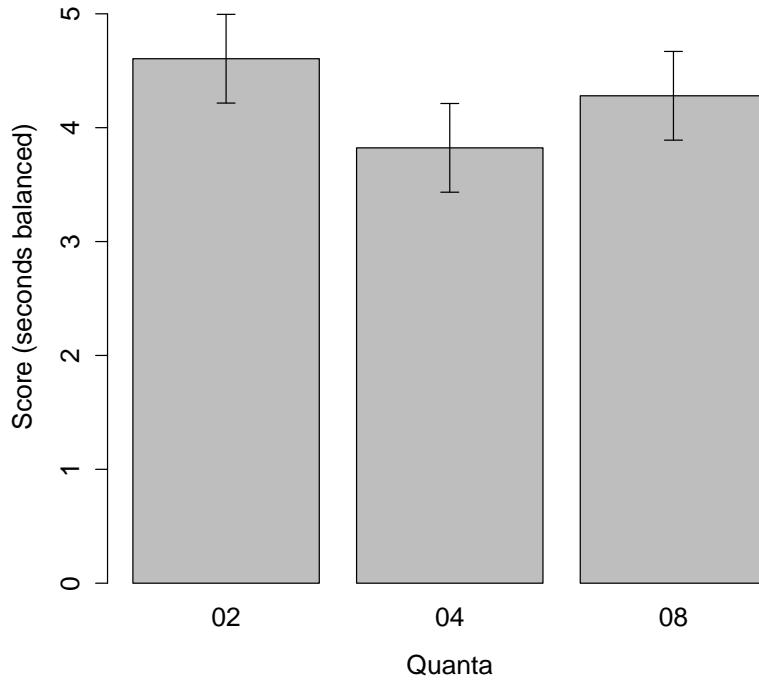
There is no statistically significant difference at the 5% level between any of the controllers. However, the performance of all of the controllers is very poor, and it seems the integrate-and-fire model is not a good fit to this robot control task. Analysis of the “model” factor has already shown that the integrate-and-fire neuron model was the worst performing of them all, so it would not be recommended to use this model anyway. Further research is necessary to determine why the integrate-and-fire model performs poorly on this task — it is a standard accepted model of a biological neuron, so hypothetically it should be capable of generating dynamic behaviour. The SRM spiking model did perform well, so the fact that this is a spiking model does not in itself explain the results.

10.7.11.6 Spike Response Model



There was no significant difference between any of the controllers, including floating-point, at the 5% level. The performance of this model is actually quite good. It is a spiking neuron model, unlike the other successful evolved controllers, which have continuous dynamics. The success of this model shows that the fixed function that converts spikes to motor signals (see section 8.12) — or rather, cheats and examines the internal activity state of the neuron, actually works. It would be an interesting extension of this research to attempt to co-evolve spike train encoding and decoding functions from scratch (the research closest to this idea so far was probably the optimisation of coefficients for a decode function in [87], however this is quite a different concept from evolving the whole decode function from scratch).

10.7.11.7 Logical model



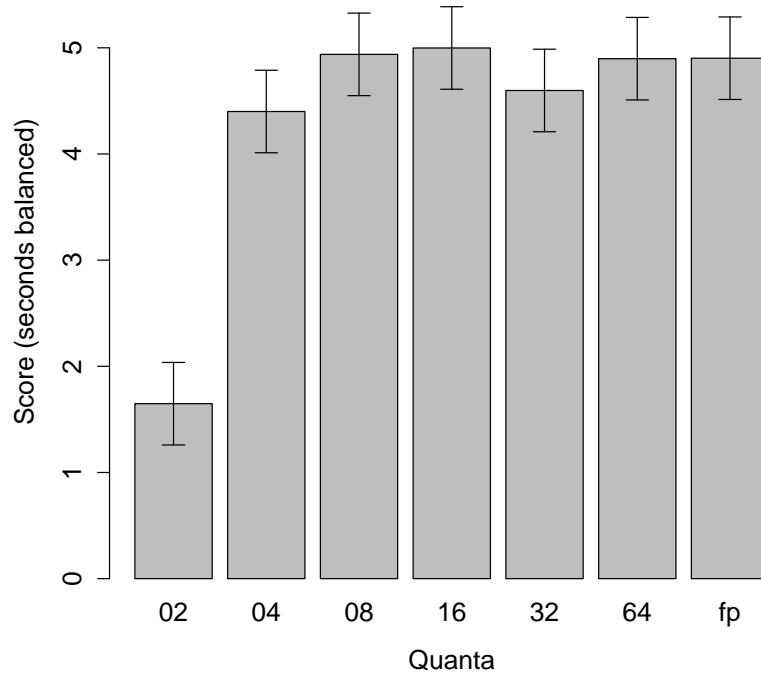
There is no significant difference at the 5% level between controllers with different quantisation levels (multi-value logic). The p value of two quanta versus four quanta is 0.056. The larger quanta cases are missing, as the complexity of the required simulations is computationally intractable (the implementation used lookup tables to store each logic function, but an alternative implementation could do things differently to enable more quanta states, so this is not some fundamental limit).

Controllers based on multi-value logic are very rare in the robotics world, and the author is not aware of any examples in the field of evolutionary robotics other than the digital circuits evolved by Thompson [441]. The main problem with arbitrarily evolved circuits is that they tend to become stuck in a point attractor rather than oscillate or generate useful computational behaviours. In biology, the multi-value generalised logical network model (see section 4.6) is sometimes used to model genetic regulatory networks. Since genetic regulatory networks display complex behaviour (construction of the morphology of living creatures), there is reason to suspect that multi-value logic networks may also be capable of similar complex behaviour.

The results here show that, for this evolved control task, there was no difference between the performance of evolved logical circuits with different quanta levels. The

performance was also quite reasonable, suggesting that a binary logical network could be used to successfully balance a pole.

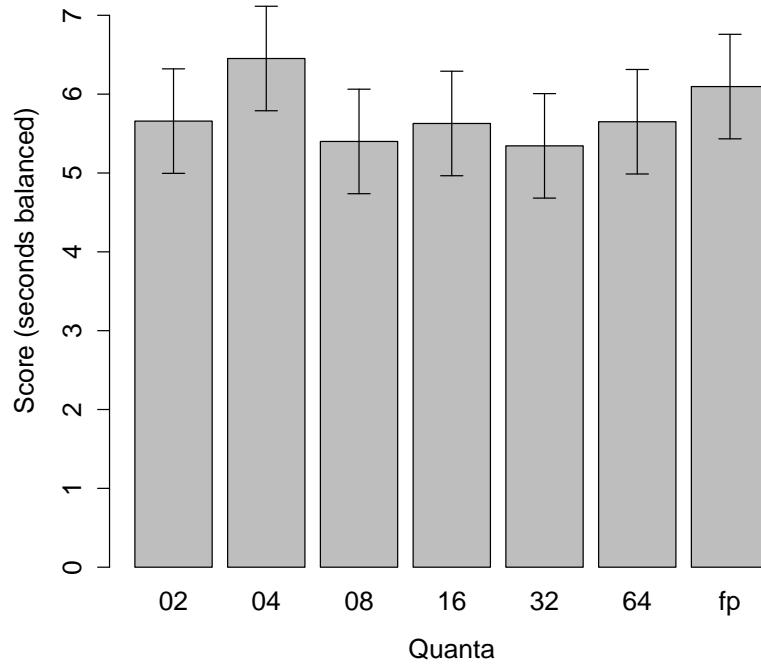
10.7.11.8 Sine model



Two state (binary) controllers are significantly worse than all the rest. There is no difference between any of the others, including the floating-point controller.

The surprising success of the sine wave model (it was ranked joint fourth) indicates something about the nature of the evaluated fitness task. The sine neuron disregards all of its inputs and produces a sine wave output signal, with evolved phase offset, frequency and amplitude. The fact that the pole balancing task can be effectively solved by a strategy of oscillating backwards and forwards suggests that other neuron models of similar performance may similarly be disregarding the input signal and producing only a regular oscillating output. However, it also shows that oscillation is not the best strategy for the pole balancing task, as there were other evolved models (sigmoid and SRM) and the non-evolved LQR controller, which all performed better.

10.7.11.9 Linear Quadratic Regulator model



There is no significant difference at the 5% level between any of the controllers, including floating-point. These controllers all perform well, and the equivalence of the floating-point implementation to all the others means that, on this task, floating-point arithmetic based controllers can be replaced with evolved binary controllers with no resulting degradation in controller performance. This is an interesting result, as linear quadratic regulators with floating-point arithmetic are well understood and widely used. This result shows that, for applications where it matters, reduced precision arithmetic and controllers should be considered, as performance may be comparable, and the implementation will be simpler and consume less power.

10.8 Summary

This chapter has explored whether evolved quantised neural networks can be used for the pole balancing task, and if so, how degraded their performance will be when compared to floating-point networks. Several published works have shown that neural networks can be evolved to solve the pole balancing problem [163, 325, 340, 409, 479]. Without exception, these works have all used neural network models with floating-point arithmetic. The motivation for exploring whether quantised neural networks can be used instead is that quantised networks do not require floating-point arithmetic units, and hence are simpler to implement and consume less power.

A fractional factorial experiment was devised to test the hypothesis that quantised neural network controllers could be evolved to solve the pole balancing problem. The experimental design varied many parameters to determine if these parameters were significant, and if so, to see how the different levels of these parameters would affect the performance of the evolved neural controllers. ANOVA modelling showed that there were several significant factors: neuron model, number of quanta states, a combined factor representing the size of the population and number of generations to evolve, and the type of neural synchronisation. Additionally, there were several interaction factors that were significant: an interaction of neuron model and the population size/number of generations factor, an interaction of the neuron model and the type of neural synchronisation, and an interaction of the neuron model and number of quanta states.

The result of the experiment was that successful quantised pole balancing controllers were evolved. The most successful models were the sigmoid model and the spike response model. Both achieved high mean scores that were significantly better than all of the other models. The quantised versions of these models were compared to the floating-point implementations.

There was no significant difference between the mean performance of sigmoid neural networks using full floating-point arithmetic and those using quantised arithmetic with a precision of 2-bits or greater. This is a significant finding, as sigmoid controllers have been widely used in evolutionary robotics for control tasks like pole balancing. The results show that, on this particular task, a floating-point sigmoid controller can be replaced with a 2-bit quantised controller without loss of performance. This suggests that other control tasks utilising floating-point sigmoid controllers may similarly benefit.

There was no significant difference between the floating-point version of the spike

response model and any of the quantised models at any level of arithmetic precision. Again, this was an interesting finding, meaning that the highest performing spiking neural network on this task could also have its precision reduced without loss of performance.

The results showed that increasing the population size and number of generations did result in neural network controllers with significantly higher performance on this task. No upper limit on the gains available given additional computational capacity was established — it is possible that increasing the population size and number of generations above 100 would result in controllers with even higher performance.

In general, synchronous neural networks had higher performance than asynchronous ones. This is likely due to the nature of the task. Synchronous controllers are more likely to generate oscillating patterns due to the fact that a single global signal can drive global patterns. However, since the performance of the sine model controllers was not the best, we know that oscillation alone, whilst a good strategy for pole balancing, is not the best strategy. Sensed input signals are important.

To sum up: Evolutionary solutions to the pole balancing problem have in the past always relied on floating-point neural models. This chapter has described research in which quantised neural networks were successfully evolved to solve the pole balancing task. For the two best performing models, there was no discernible difference between the floating-point implementation and the quantised implementation. This means that the pole balancing task can be solved with evolved quantised models without loss of performance, which is an important finding as quantised models are simpler to implement and consume less power.

Chapter 11

Virtual creature experiments

The previous chapter compared floating-point and quantised neural controllers evolved to carry out the pole balancing task in a fixed robot morphology. This chapter will compare the performance of floating-point and quantised neural controllers on the task of generating locomoting behaviour in evolved virtual creatures.

It has been claimed that the co-evolution of morphology and control allows a more natural evolutionary path to be followed, with a greater chance of successfully evolving high fitness individuals (see section 6.14) [50, 67, 257, 321]. It has also been claimed that decentralised digital networks are capable of rhythmic pattern generation behaviour (see section 6.12) [107, 108, 364], and it has been suggested that “central pattern generators” based on continuous dynamics are responsible for motion behaviour in living creatures (see section 2.4) [195, 270, 279, 307]. This chapter will therefore extend the experiments of the previous chapter, evolving both control and morphology of virtual creatures, and again comparing the performance of continuous and quantised neural networks. The experiments in this chapter involve the evolution of genotypes with combined morphology and neural network control systems as already described in chapter 8.

11.1 Introduction

The hypothesis investigated in this chapter is that quantised neural networks can be evolved to generate rhythmic patterns which can be used to drive robot locomotion, and that the performance of these networks on the locomotion task may be comparable to that of floating-point networks, which are more widely used in evolutionary robotics. Most evolutionary robotics research relies on complex neural models for con-

trol. There is often a basic, unchallenged assumption that a floating-point implementation should be used, and little consideration is given to using simpler neural networks. This is partly due to the fact that evolutionary robotics experiments often make use of software based physics simulations carried out on PCs, which have a mains power supply and CPU with integrated high-performance floating-point vector arithmetic units. However, for real autonomous robots energy usage is a major concern, and simpler neural control models would lead to substantial savings in power consumption.

Since we have no concept of how to manually design dynamic neural networks for complex control task, genetic algorithms will be used to evolve all of the functional parameters of the digital neural networks. The networks will be linked to a 3D physics simulation system for the purposes of determining evolutionary fitness (the complete architecture is described in chapter 8). The genotype will encode both morphology and control, in order to increase the probability of successfully evolving working controllers, and to explore whether evolved quantised neural networks can successfully generate locomoting behaviour in co-evolved morphologies.

We wish to show that simple, quantised neural controllers are suitable for the robot control task. The performance of various kinds of digital control network will be compared to that of floating-point neuron models. It is already known that floating-point models can be successfully evolved to generate locomoting behaviour in co-evolved morphologies (see section 6.14). The hypothesis of this chapter is that quantised models can also be used to generate locomoting behaviour.

Using genetic algorithms to evolve robot morphologies and neural network controllers presents some problems. Quite often in previously published research various parameters of the genetic algorithm will be set to particular constant values without any explanation as to why these values were chosen. Parameters such as the population size, number of generations, and number of neurons in a neural network, are often stated as fixed, but no justification is given as to why or how the fixed value was chosen. This chapter will therefore attempt to establish whether some of these factors are significant in determining the performance of the evolved robots. However, it is not feasible to investigate all possible factors and their many interactions, and for this reason many of the parameters will remain fixed (such as using an elitist genetic algorithm, and a generational genetic algorithm).

11.2 Task

The creature morphology, morphogenesis, neurogenesis, and neuron models were all as documented in chapter 8. An elitist genetic algorithm was used, with the top 5% of the population surviving intact to the next generation and being used as parents to generate children. Simulations were fixed to last exactly 30 simulated seconds. Mutation was the sole genotype reproduction operator. The mutation probability for parameters of the genome was set at 0.1, meaning that every individual parameter had a 10% chance of being mutated during genotype reproduction. Each cylinder of a morphology had a neural network that was evolved using the same parameters as in the pole balancing experiment. The robot architecture, evolvable parameters and evolutionary process have already been documented in chapter 8.

There are many factors which affect the evolutionary process. Some of these are: the neuron model, whether the neuron model is continuous, or if discrete the number of quanta states, the topology of control networks, the number of neurons, the type of genetic algorithm used (generational or steady state), the mutation probability (usually a fixed constant, though sometimes varied dynamically), the type of mutation (uniform or Gaussian), number of generations, size of population, the fitness function (including simulation time), the updating scheme of the neural networks (synchronous, asynchronous, or other). These factors will be explored to see if varying them significantly affects the performance of the resulting evolved controllers on the locomoting fitness task.

11.3 Fitness function

Locomotion is a common task for evolved robots and their controllers. Typically the fitness function used will reward motion in one particular direction, rather than overall motion, as it is usually assumed that the aim of directed locomotion is to get to some particular place quickly and reliably rather than just randomly wandering. Sometimes other behaviours, like turning, are evolved once forwards locomotion is working.

The fitness function used here measured locomotion along the x -axis. The bodies of the evolved robots consisted of several cylindrical parts, each of which had its own weight and centre of mass. This complicates the fitness function; do we want to measure the displacement of the composite robot by averaging some metric of each individual cylinder, thus giving a simple global measure of overall distance moved? Or

should the fitness function treat each cylinder individually, attempting to ensure that all cylinders are moved some minimum distance? The fitness function shapes the evolutionary search, so attempting to optimise simple parameters can often have unintended consequences.

Pilot runs were carried out to explore how the evolutionary algorithm optimised evolved morphologies and controllers in the presence of different fitness functions. All functions aimed to measure some metric of x -axis locomotion. One function simply calculated the mean displacement of all cylinders along the x -axis. The result of this was that the genetic algorithm evolved increasingly taller robots that would be standing at 0 seconds, but would then immediately fall along the x -axis, producing a high fitness score (see figure 11.1). This was effectively an evolutionary dead end, with subsequent generations producing taller individuals, and increasing the number of “high” cylinders, as these would attain the greatest displacement once the robot had fallen.

A proposed fix for this behaviour was to use a fitness function that measured the displacement of the cylinder closest to $x = 0$ after 30 seconds. Again, pilot runs showed this could be exploited by the evolutionary system, with the early population becoming dominated by a simple 2-cylinder hinged robot with a constant value signal wired to its motor. Once the simulation began, the robot would push against the ground, producing a leap along the x -axis, but once it had landed there would be no further movement (see figure 11.2).

After some testing, a simple fitness function was devised that measured locomotion along the x -axis but did not appear to be as exploitable. The function was \min_x/t where \min_x was the displacement of the cylinder closest to $x = 0$, and t was the current simulation time. This function measures the *velocity of the slowest cylinder*. It encourages movement along the x -axis, but also ensures that all body parts must move. Pilot runs showed that this fitness function was more likely to lead to continuous locomotion than the others, and there did not appear to be an obvious way for the evolutionary algorithm to exploit it towards dead end non-locomoting solutions. One advantage of using the overall velocity as a measure of locomotion is that it is a time invariant metric; the results from simulations with different timescales are directly comparable. Each simulation would last exactly 30 seconds.

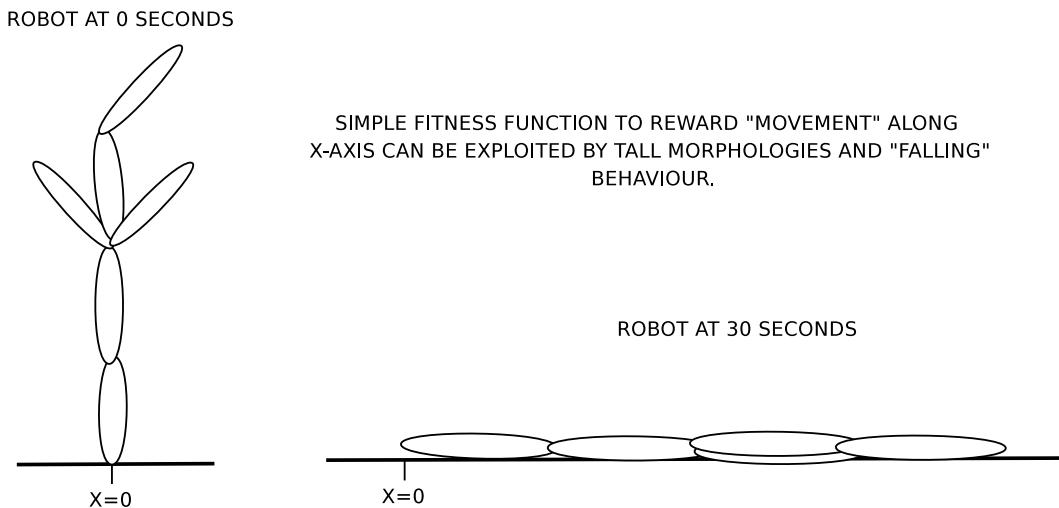


Figure 11.1: A fitness function attempts to reward locomoting behaviour along the x -axis. Using a simple measure such as the mean displacement of cylinders between $t = 0$ and $t = 30$ can lead to robots that exploit this function by growing increasingly taller, and during simulation just fall to one side. No real locomotion occurs, but the robot gains a high fitness score.

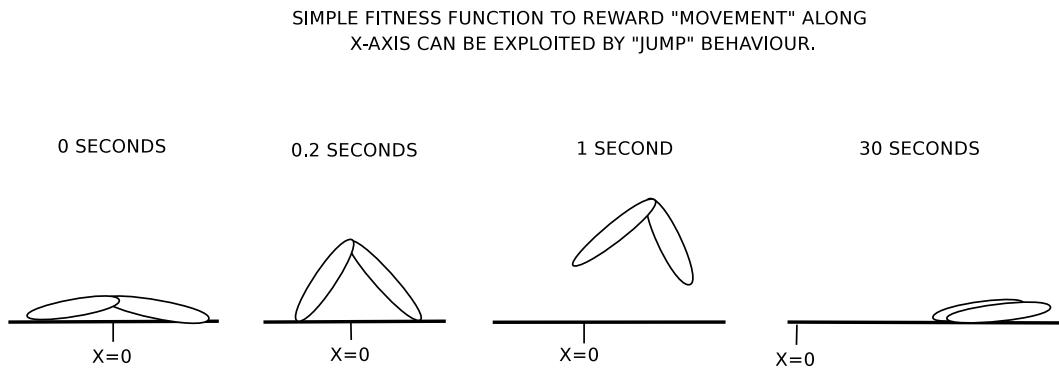


Figure 11.2: A fitness function attempts to reward locomoting behaviour along the x -axis. Using a simple measure such as the final displacement of the robot at $t = 30$ can lead to simple robots that exploit this function by constantly activating a motor on a single joint to drive the two cylinders into the ground, producing an upward motion. The robot will only leap once, but gains a high fitness, effectively creating an evolutionary dead end with no real locomotion.

FITNESS FUNCTION TO ENCOURAGE SUSTAINED
MOVEMENT OF ALL CYLINDERS.

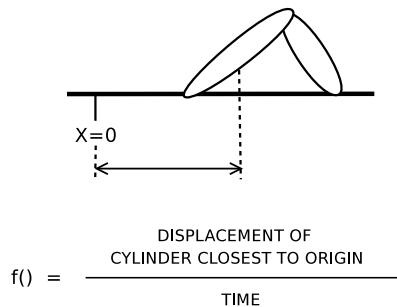


Figure 11.3: *The final fitness function effectively measures the mean x-axis velocity of the slowest cylinder. It encourages movement of all cylinders along the x-axis, and is a time invariant measure.*

11.4 Experimental design

The experimental design was the same as in the pole balancing experiment (see section 10.5). There were no extra factors to vary in the evolution of virtual creatures. The existing factors and levels were also sufficient for the purpose of qualitative evaluation of the different neural models and quantisation combinations. In total 533 replicates were evaluated, requiring 6.6 million individual fitness evaluations, a total run time of approximately two weeks on a cluster of 24 3GHz dual-core PCs.

Pilot runs of the genetic algorithm showed that the computational requirements could be reduced without any significant change in the evolvability or final results. The genetic algorithm ran three separate simulations as part of the fitness evaluation of an individual. This policy had been introduced following stability problems simulating the physics of arbitrary evolved morphologies; unstable points in the simulation tended to gain huge amounts of energy, resulting in unrealistic movement leading to undesirable high fitness scores. After substantial work on fixing the physics, a pilot run showed no difference between results from a fitness function using a single simulation versus one using a triple simulation. Removing the two extraneous simulations reduced the required computational time to roughly one third, although other overheads,

such as database latency and network bandwidth, meant that the final saving was not as large.

Pilot runs also showed that the choice of mutation operator (Gaussian or uniform) and mutation rate could substantially affect evolutionary behaviour. Search can be characterised as two phases: an initial phase across a massive space looking for a basic concept, in which large leaps are necessary, and a subsequent optimisation phase in which that basic concept is refined, requiring short steps. If the mutation rate is low, or Gaussian mutation is used (in which values are chosen probabilistically from a Gaussian distribution centered around the current value), and if there is no workable solution in the initial generation, then the genetic algorithm is unlikely to find one, and final results will be low scoring. In contrast, if mutation is uniform and with high probability, then the genetic algorithm can lose good solutions, since they are highly likely to be mutated. Using a more adaptive genetic algorithm, which can dynamically vary mutator and mutation rate, may produce more optimal behaviour.

11.5 Reproducibility

As noted in the previous chapter, the complete experimental setup described here consists of thousands of lines of source code. Small changes in the experimental setup may cause changes in the observed data, so the experimental setup must be documented at a very low level. It is not practical to include an English language or pseudo-code description at this level within the thesis itself, so to aid in openness and experimental reproducibility the exact source code and scripts used to run this experiment will be published along with this thesis. Section 1.3 contains some more commentary on the reproducibility of scientific simulations and the importance of this to the scientific process.

11.6 Results

11.6.1 ANOVA modelling

The process of ANOVA modelling was the same as described for the previous set of experiments in section 10.7.1. The null hypothesis was that varying the levels of each factor, or combination of factors, would not significantly affect locomoting performance of the best individual in the final evolved generation. The significance level

used was 5%.

The residuals of the *ANOVA* model were tested for normality as described for the previous set of experiments. Again, the residuals showed some small signs of non-normality. Minor deviations from normality are to be expected for real experimental data, and the statistical analysis is robust to deviations (see section 10.7.1 for more details). Figure 11.4 shows the plotted residuals for the generated *ANOVA* model. The residuals were initially approximately normal distributed. Using a log transform made the residuals more normal, but made no difference to the *ANOVA* model — the same set of factors were deemed significant (or not) at the 5% level.

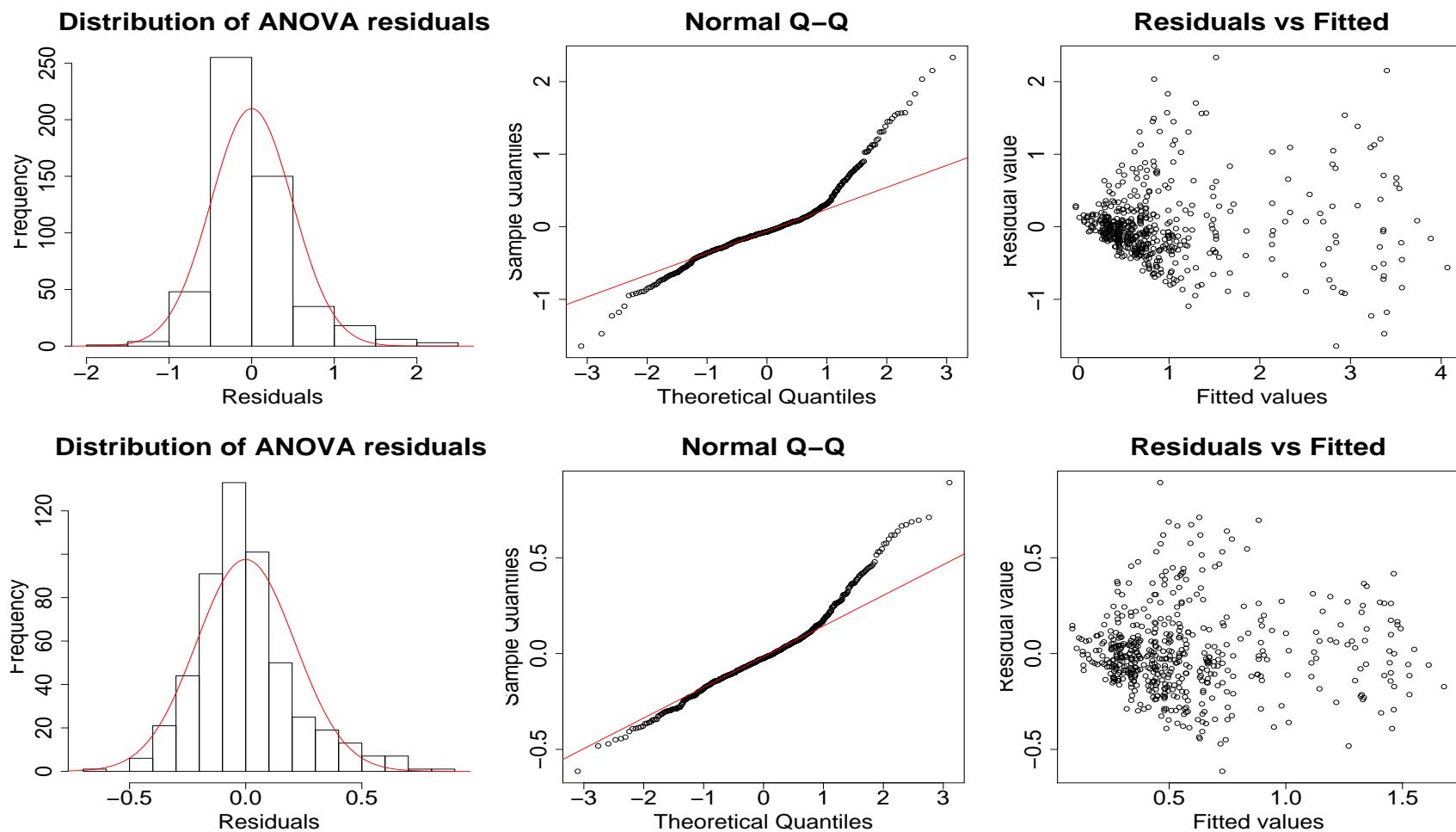


Figure 11.4: ANOVA residuals plots of the original data (top) and log transformed data (bottom). The original distribution is reasonably normal to start with. The log transformed data is more normal (bottom left), displays straighter Normal Q-Q plot (bottom centre) and shows less clustering against fitted values (bottom right). However, the fact that these differences are small, combined with the reasonable normal-ness of the original data and robustness to non-normality of the statistical tests meant that the set of significant factors remained the same.

11.6.2 ANOVA results

The method of analysis of the results was the same as described for the previous set of experiments in section 10.7.2. The significance level was 5%. The ANOVA model showed that several factors were statistically significant at the 5% level:

Factor	Significance (p)
model	$\leq 2.2\text{e-}16$
genpop	2.130e-08
timing	1.384e-08
model:timing	7.265e-07
model:q	0.02072

(the colon character denotes an interaction between two factors)

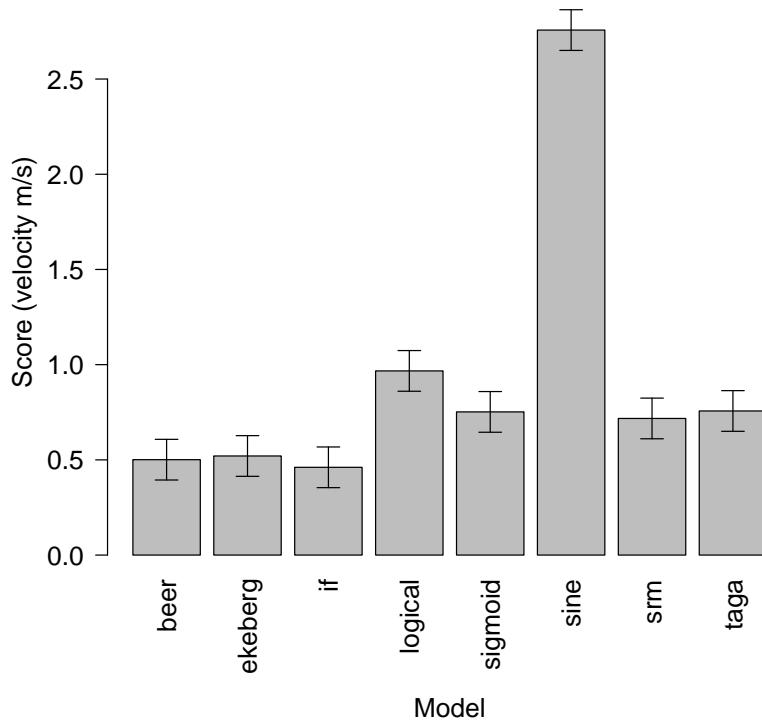
All of the other factors and possible combinations of factors were found to be not significant at the 5% level.

The following sections show various plots comparing the performance of different levels of these factors. All results are based on the overall velocity along the x -axis achieved by the best individual in a population at the end of an evolutionary run. The particular makeup of the rest of the population is not considered, as the purpose of evolving a solution is usually to find and use the best and disregard the rest.

11.6.3 “Least Significant Difference” plots

The process and explanation of the “Least Significant Difference” plots is the same as described for the previous set of experiments in section 10.7.3.

11.6.4 Factor: Model

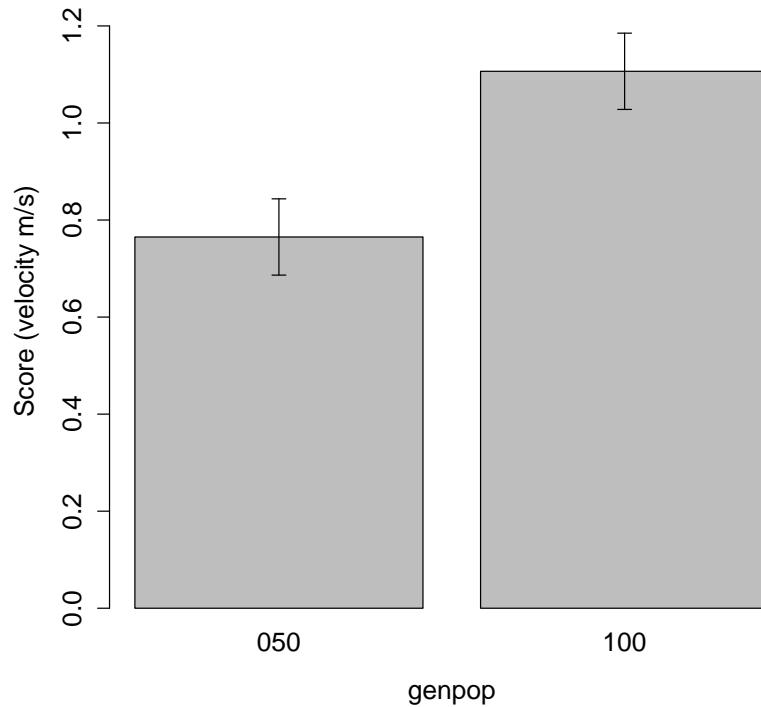


There are statistically significant deviations at the 5% level between several different models. All of the neuron models successfully generated locomoting behaviour. The sine wave model was clearly superior to all others. The logical model was next; at the 5% level there was no significant difference between it and the sigmoid model ($p = 0.068$) or the Taga model ($p = 0.079$), but there was a significant difference when compared to the SRM model ($p = 0.036$). There was no significant difference at the 5% level between Beer's model, the Ekeberg model, or the integrate-and-fire model, which were collectively the worst.

The sine wave neuron ignores all of its inputs and just generates a sine wave output signal. It has evolvable parameters of amplitude, phase offset and frequency. Given this simplicity, it may be surprising that it scores so much more highly than other more complex and more adaptable neuron models, especially since these neuron models are more widely used in evolutionary robotics for tasks like locomotion control. The success of the sine wave model shows that sensory input is not so important for the basic locomotion task, but the ability to easily generate cyclic repeating activity patterns is. The other neuron models can generate oscillating patterns, but the parameter space of the neuron in which it will oscillate is smaller, and some models require multiple

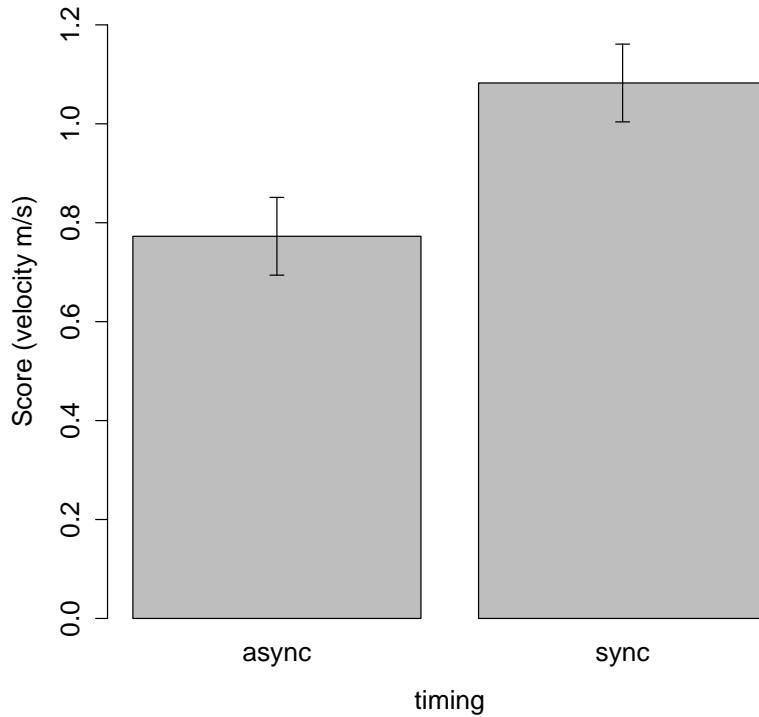
neurons to be linked together in specific configurations before oscillation will occur. Oscillating motion in sine neurons is guaranteed, whilst oscillating motion for other neurons must be discovered, and is hence less likely to occur.

11.6.5 Factor: Number of generations and population size



There is a statistically significant deviation at the 5% level between evolutionary runs with a population size of 50 evolved for 50 generations versus those with a population of 100 evolved for 100 generations. Note that we are using a combined factor here as the real underlying issue is one of computational power — the effect of varying either the population size or the number of generations is actually to vary the number of fitness evaluations carried out. These results suggest that greater computational resources may be utilised to evolve controllers with better performance. There is an open question as to what the best population size and number of generations is in evolutionary robotics, and whether there are any general principles that apply to the genome encoding of creatures, or to the phenotype model or fitness task, that would enable the best values to be estimated before the evolutionary process is carried out.

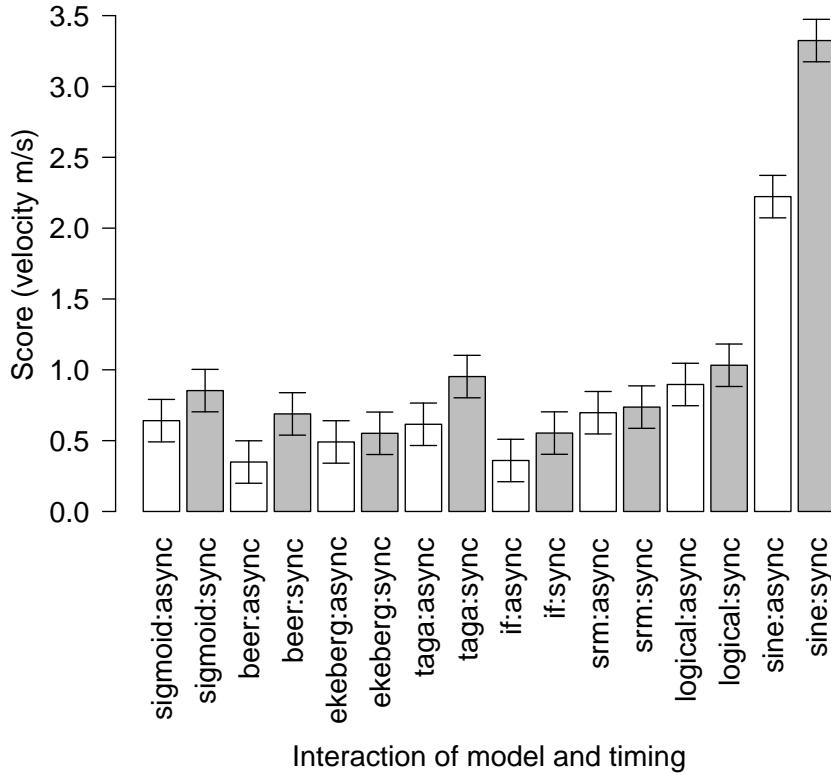
11.6.6 Factor: Timing



Controllers with synchronous timing perform significantly better than those with asynchronous timing. Synchronous timing is completely predictable, whereas asynchronous timing is not, and this may aid controllers in producing stable and repeatable patterns of activity. It is known that synchronous nk networks and cellular automata are more likely to display rhythmic patterns of activity than their asynchronous counterparts [107, 108], and so solutions are easier for the genetic algorithm to discover. The result also suggest that synchronicity provides a global timing which can be used to coordinate the movement of different joints.

This result leaves open the question as to whether or not there may be other kinds of timing model that would work better. Global synchronicity is undesirable and it is computationally more intensive (consuming more power), and requires complex wiring to route the single timing signal to every neuron in the network (which may not even be possible if we require fast neuron response but have a large, high latency network). It would be useful to investigate and compare the many different hybrid timing models, which provide synchronisation between individual cells or set of cells, but not globally.

11.6.7 Factor: Interaction of neuron model and timing



This factor represents the interaction between the neuron model and whether the timing is synchronous or asynchronous. We already know that the *timing* factor is significant, and that on average (across all models) synchronous controllers outperformed asynchronous ones. What this interaction factor shows is whether synchronous controllers outperformed asynchronous controllers for individual neuron models. There is a visible trend that synchronous is better than asynchronous for all models, although the difference is only statistically significant at the 5% level for the Beer, sine, and Taga models.

The large difference between the two models of sine neuron may seem surprising given that the sine neuron ignores all inputs and hence synchronisation does not seem so important. The asynchronous updating scheme carries out a number of updates equal to the number of neurons in the network, so has the same “computational cost” as the synchronous one, but the neuron to update is selecting randomly. This is a purely asynchronous approach, where there is no timing at all, and hence it is possible for a single neuron to be updated more than once, whilst another may not be updated

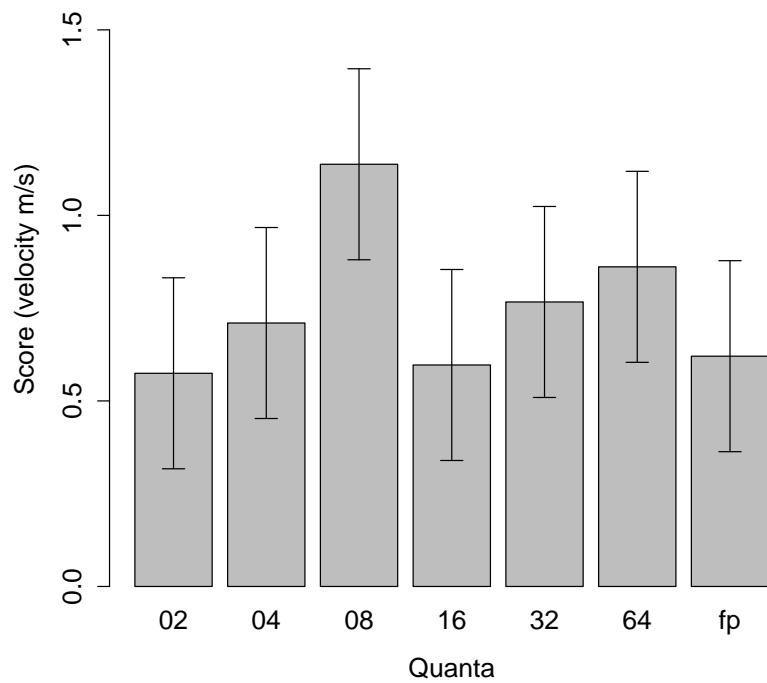
at all. This results in desynchronisation of the signals driving different joints, which negatively affects the ability to generate coordinated locomoting behaviour.

The fact that there are significant differences between the asynchronous and synchronous versions of some neuron models but not others is interesting, as it suggests that the dynamical behaviour of some models is more robust than others to completely desynchronisation operation.

11.6.8 Factor: Interaction of neuron model and quantisation

The performance of each quantised neuron model is individually plotted against the number of quanta states (corresponding to the arithmetic precision of the model). The null hypothesis is that there is no difference between the different quanta levels. The alternative hypothesis is that the means of two or more levels are different. The significance level used for comparison is 5%.

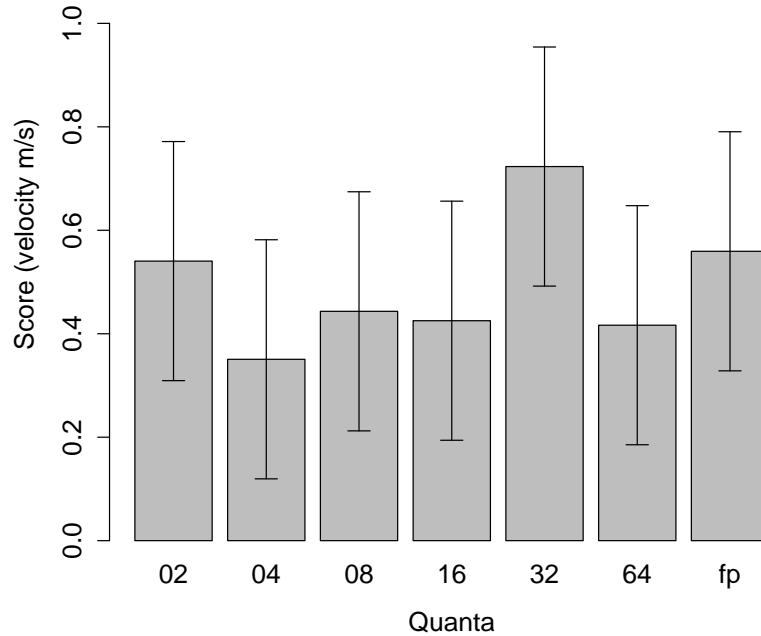
11.6.8.1 Sigmoid model



The eight quanta case is significantly different from the floating-point case at the 5% level. This is either a result of some synergy of the 8 quanta sigmoid model, or a

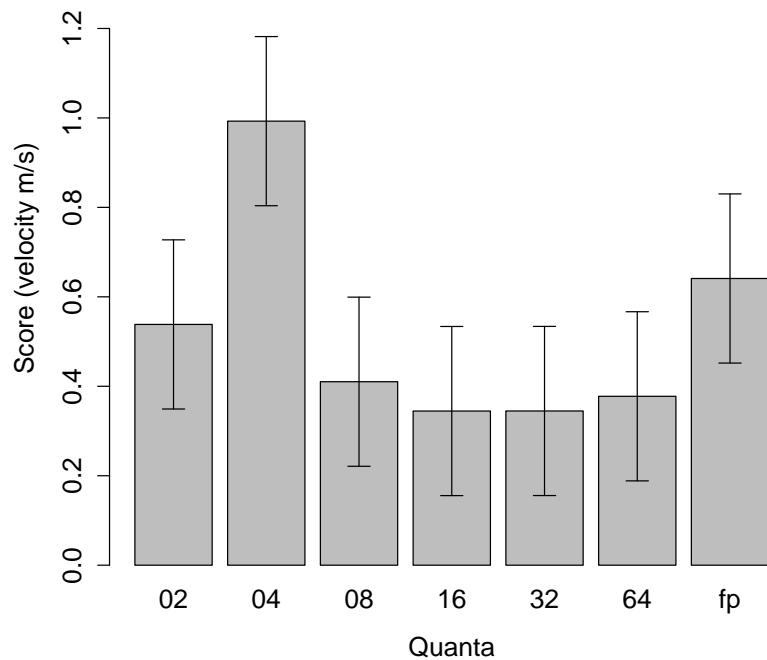
statistical anomaly. Apart from that unusual result, there is no difference between any of the different quanta controllers at the 5% level.

11.6.8.2 Beer's CTRNN model



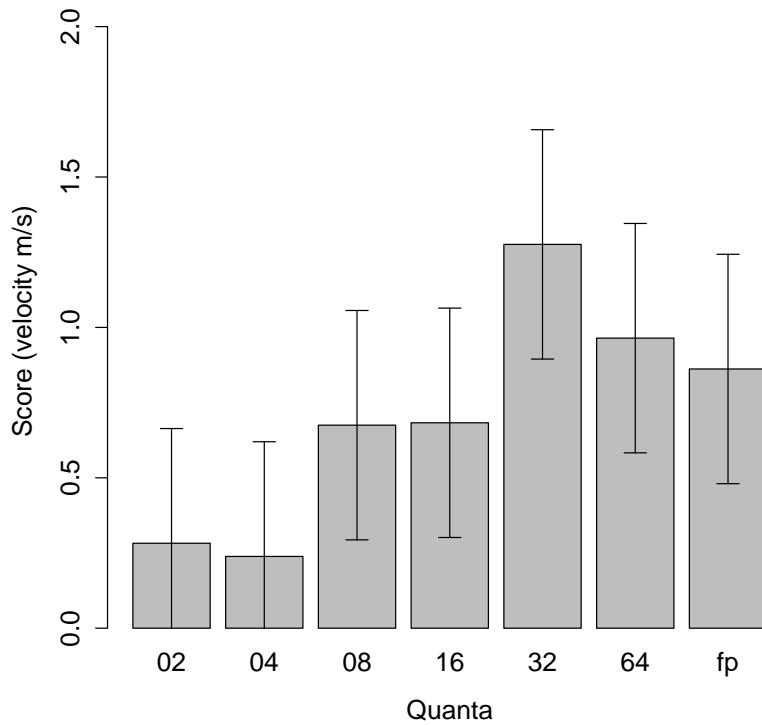
There is no statistically significant deviation at the 5% level between the mean values for different quanta. This is an interesting result for Beer's model, as the floating-point implementation has been widely used in evolutionary robotics research. The fact that it is equal in performance on the locomotion task to a simple 1-bit model suggests that the floating-point model is more complex than required for many of the applications to which it has been put.

11.6.8.3 Ekeberg model



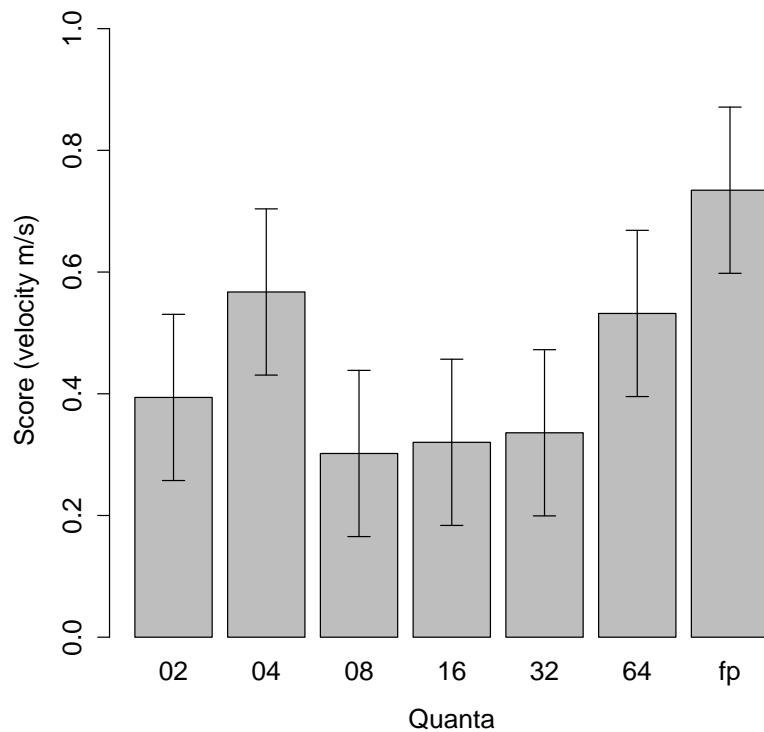
The four quanta case is significantly higher at the 5% level than the other quanta levels, but not higher than the floating-point case. This unusual result suggests that there is some interaction between the quantisation and the Ekeberg model when the number of quanta is four. Perhaps it makes the model more likely to oscillate. There is no significant difference between the other levels, suggesting that simple binary Ekeberg models perform as well as floating-point models on the evolutionary locomotion task.

11.6.8.4 Taga model



The 32 quanta model is significantly better at the 5% level than the 2 and 4 quanta models. The difference between the other models is not significant. In particular, there is no difference between the floating-point model and any of the others, which suggests that the floating point-model may be replaced with quantised models without degradation in performance on the locomotion task.

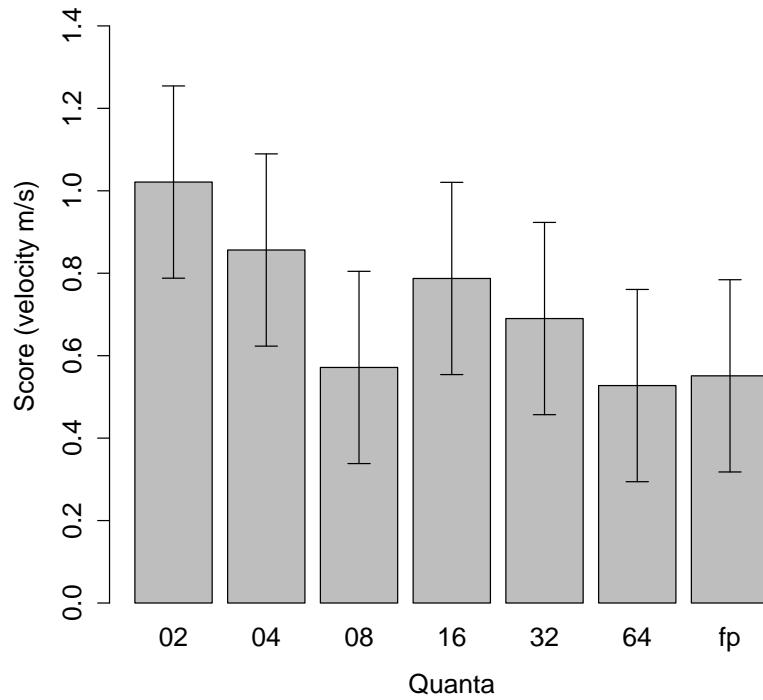
11.6.8.5 Integrate-and-fire model



The floating-point case here is statistically superior at the 5% level to four of the quantised cases. There was no statistical difference between the four and sixty four quanta cases, and the floating-point case.

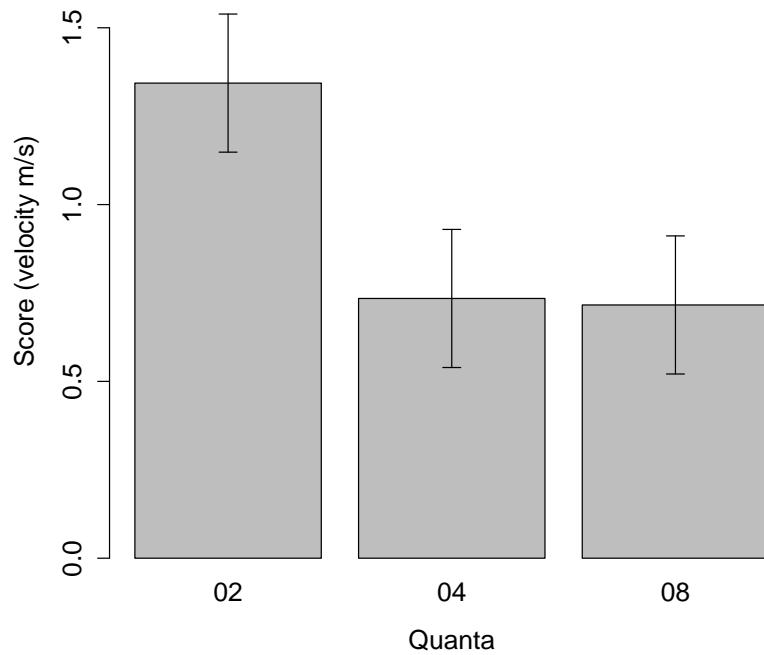
There is no difference between the floating-point model and the 4 quanta and 64 quanta models. The high mean for the four quanta controllers may be an anomaly, or there may be something peculiar about this model that encourages pattern generating behaviour. There is no statistical difference at the 5% level between the 64 quanta and floating-point model, suggesting that quantisation in itself has not negatively affected performance.

11.6.8.6 Spike response model



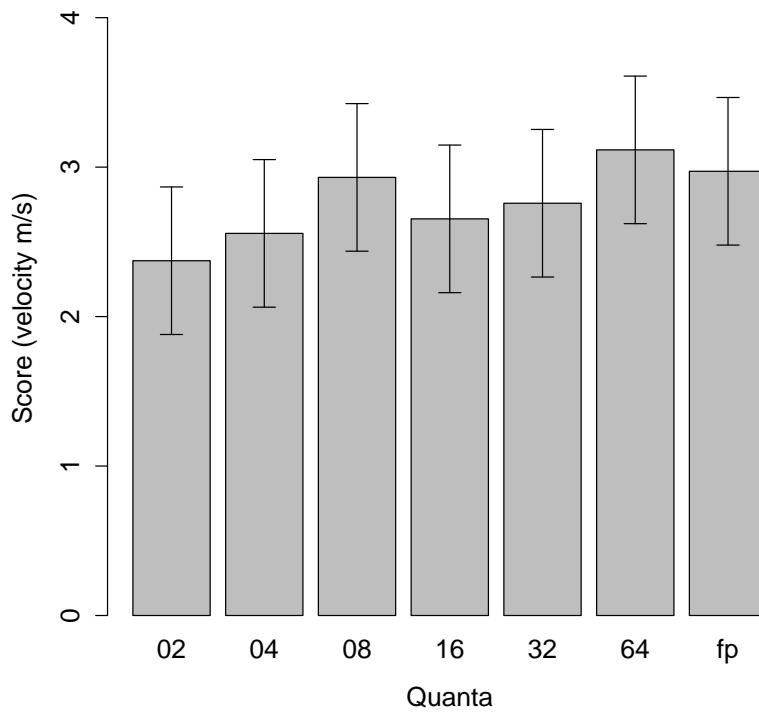
Interestingly, these results show a trend towards declining performance as the model becomes more complex. The two quanta binary model is the highest performing, and is significantly better than the floating-point model at the 5% level ($p = 0.043$).

Again, it is possibly the case here that a two quanta neuron is more likely to oscillate, and that this results in better performance on the locomotion task.

11.6.8.7 Logical model

The two quanta case is statistically superior at the 5% level to the four or eight quanta cases. The larger quanta cases are missing, as the complexity of the required simulations is computationally intractable. The superior performance of the two quanta case is interesting, and probably a result of the fact that with only two quanta states oscillation between them is going to result in high amplitude signal oscillation.

11.6.8.8 Sine model



There is no statistically significant difference between any of the quantisation levels, or even the floating-point model, suggesting that a simple binary oscillator (2 quanta) neuron is equivalent to the most complex floating-point model. Several evolutionary robotics research projects have used floating-point sine waves to generate locomoting behaviour. For this kind of project, the sine wave can be optimised by pre-computing the signal values and storing them in a lookup table. The results here suggest that an even simpler and just as effective optimisation would be to replace the sine wave neuron with a binary oscillator.

11.7 Example evolved control

The hypothesis is that quantised neural networks can be evolved to generate rhythmic patterns which can be used to drive robot locomotion. Successful locomotion control consists of a discrete or continuous dynamical system that can generate internal oscillations and present these as outputs to drive motors. The neural network and signal dynamics of quantised controllers of each neuron model are presented here. The controllers chosen to be presented here were not necessarily the best performing, but had networks and signal traces that were relatively clean and with obvious rhythmic dynamics. The aim of this section of the thesis is to demonstrate some quantised rhythmic dynamics that generate successful locomotion for each neuron model, and to give the reader an idea of how these signals and networks look. One interesting observation is that many of the controllers work by generating oscillating outputs connected to joint motors, and that the most successful model, the sine neuron, does only this, suggesting that sensory input is not so important for the locomotion task.

11.7.1 An explanation of these graphs

For each example robot two graphs will be presented. One shows part of the nervous system of the complete robot taken from a single cylinder. The other graph shows some of the neural signals recorded from this section of the nervous system. The complete robot nervous system will not be shown, as it is lengthy and not so informative. For the same reason, the neural signals graph for each example will be limited to a single page and may not show the output of all neurons. The aim of this section of the thesis is to show some of the evolved dynamics of the quantised neuron models, and to show that they generate regular rhythmic patterns. This thesis is not intended to convey an in-depth examination of the dynamical systems that these nervous systems form.

11.7.1.1 Nervous system plot

The nervous system plot shows the nervous system, or part of the nervous system, of the robot. Each cylinder of the robot's morphology is plotted as a box, with edges between these boxes showing the tree structure of the morphology. These edges are rendered as dashed lines, but are often so short that only the arrow-head is visible. The section of the nervous system associated with each cylinder is plotted inside a box as a graph of neurons, sensors, and motors. Inter-cylinder connections may exist between

these graphs.

Edges between the nodes represent neural connections. The edges are coloured green for excitatory connections and red for inhibitory connections. For neural network models with weighted connections, the brightness of an edge represents how strong the connection weight is — bright green represents a high positive weight, light green a low positive weight, bright red a high inhibitory weight, and light red a low inhibitory weight. The colour is a linear scaling of the value domain of connection weights onto red or green pixel intensity. Note that one effect of this colour coding scheme is that as the weight of an edge approaches 0 the edge becomes lighter to the point of not being visible. This is probably undesirable, as the viewer of the graph is interested in seeing the complete topology, so to prevent this happening a minimum intensity level is enforced for all edges (the exact value is 20% of full brightness). Unweighted connections, such as those used by logical networks, are rendered as black edges.

Each box is labelled with a string in the format “bpX ([root or joint_type], neuron_type)”. “bp” stands for “body part”. The digit X is a number that is unique for each cylinder. This is followed by some text in brackets. The text “root” indicates that this cylinder is the root cylinder, which has no joint and hence no motors (joint sensors are still present but always return 0). If the cylinder is not the root cylinder, then the text states the joint type (hinge, universal or ball) that connects it to its parent. The “neuron_type” is the neuron type used in the neural network for this cylinder (Beer, Sigmoid, Ekeberg, Taga, If, Srm, Logical or Sine).

The nodes within the box are labelled to indicate a unique identifying number, the type if the node represents a sensor, and whether or not the node can be used as an input or output to connect to the networks of other cylinders. If the label begins with a digit, then it is a standard neuron (the type is already stated in the box label). Otherwise “J” indicates a joint, “M” indicates a motor, and “C” indicates a contact sensor. Joint and motor nodes have a suffix digit that indicates which axis of the joint or motor this node represents (0, 1, or 2). The suffixes “i” and “o” for standard neurons indicate that this neuron may (but does not have to) be connected as an input or output to the networks of other cylinders.

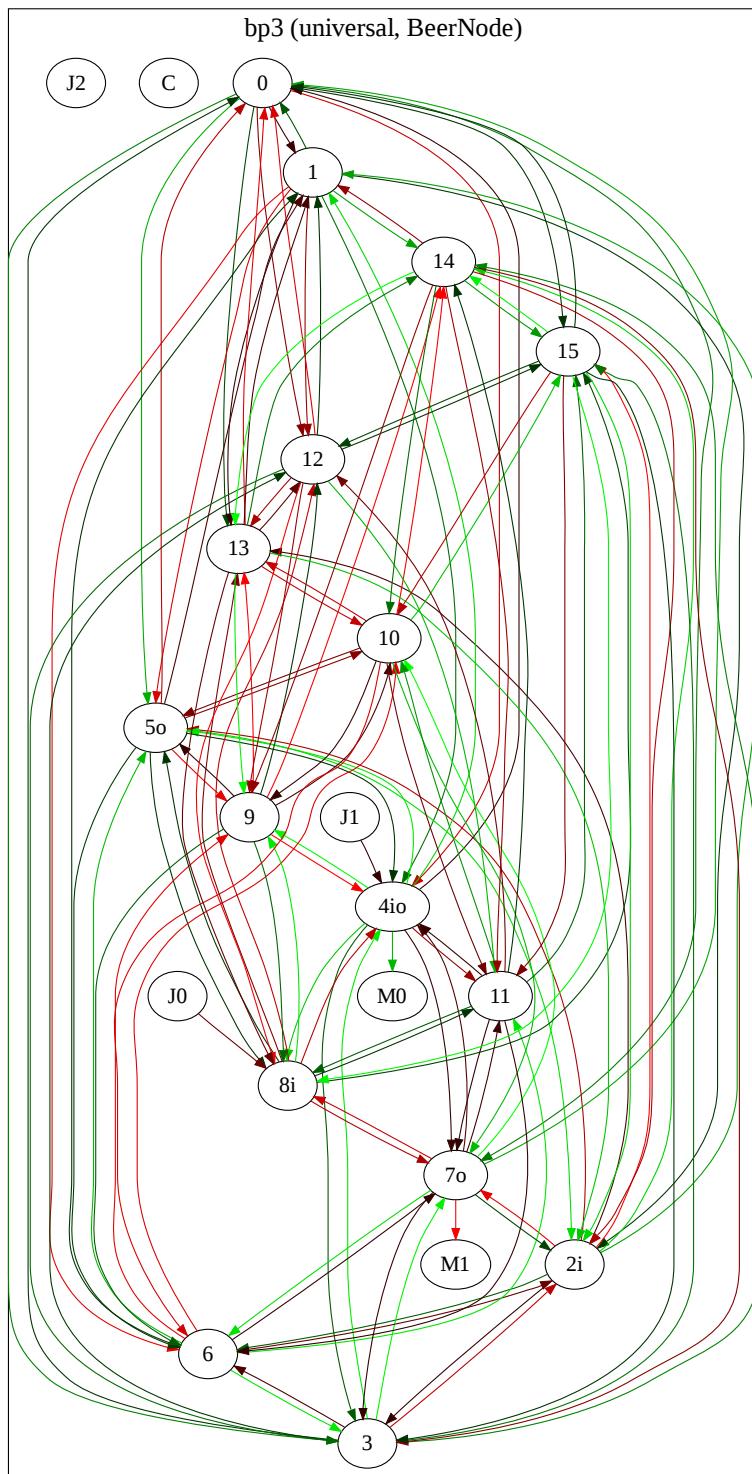
11.7.1.2 Neural signals plot

The neural signals plot shows the output signals of each neuron during simulation. There are usually many neurons, and quite often a few of them output constant values. These constant signals impart little knowledge to the reader, and so are not plotted.

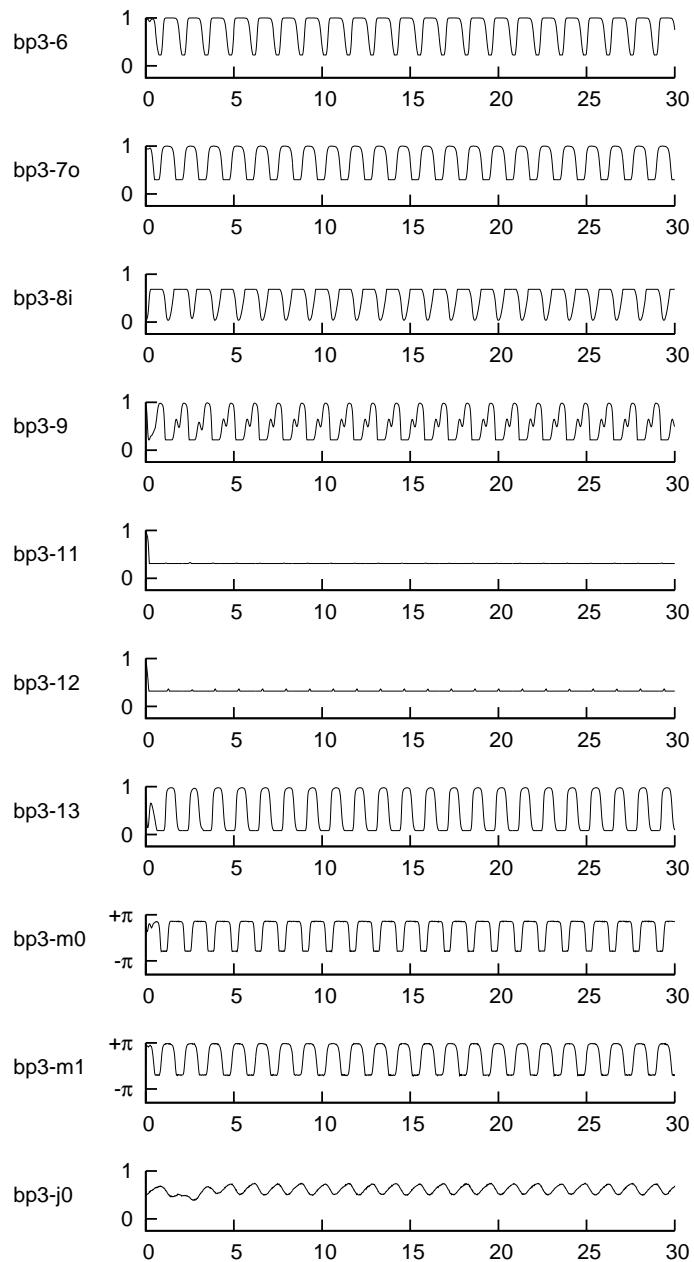
The output of a single neuron is normalised and plotted as an individual graph. The output of joint sensors is plotted in the range $-\pi..+\pi$. The x -axis shows the simulation time in seconds. Note that the x -axis is placed below the 0 point on the y -axis, which means that the signal plot is not obscured by the actual x -axis when the signal value is 0. The y -axis label shows which cylinder this network belongs to. The label is in the format “bpX-T[io]”. “X” is the number of the cylinder — each cylinder is assigned a unique number to identify it. “T” is the type of neuron: “m” for a motor, “j” for a joint angle, “c” for a ground contact sensor, or a sequence of digits for a regular neuron. The sequence of digits uniquely identifies the neuron within its network. The number can be suffixed by “i” or “o” to identify that the neuron may be used as an input or output to connect this network to the networks of neighbouring cylinders.

11.7.2 Example floating-point controller (Beer’s CTRNN model)

This is a network of Beer’s CTRNN neurons using floating-point arithmetic and synchronous updating. The robot morphology consists of four cylinders. The full network plot is large, so only one of the cylinders will be plotted along with its signals.



This plot shows the neural architecture of one cylinder. It consists of a network of Beer's CTRNN neurons, with input signals from the two joint axes (J_0, J_1) and outputs to the two axes of the joint motor (M_0, M_1). The neural network has 16 neurons arranged in a 2D geometric structure with size 3 neighbourhood. Updating was synchronous. This particular network achieved a score of 1.529 in the final population.

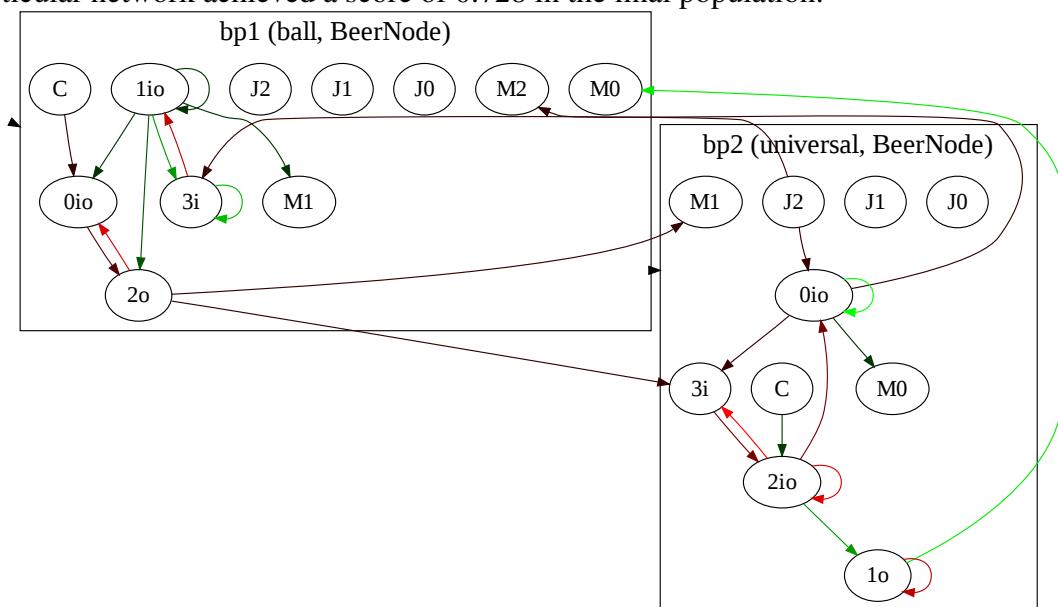


The neuron model is floating-point (not quantised), so there is a fine resolution with detailed waveforms. The two motors (labels bp3-m0 and bp3-m1) show clear repetition of a non-identical waveform. The network is initialised into a random state. In the first second the waveform is noticeably slightly different for several neurons. The important thing here is that the dynamics of the network itself attract some of the

neurons towards a limit cycle, which is manifested as a rhythmic pattern. The lowest row shows the effect of this pattern on the joint angle, which oscillates in a regular pattern after about three seconds. The joint angle has an initial transitory period before it falls into a regular rhythmic pattern.

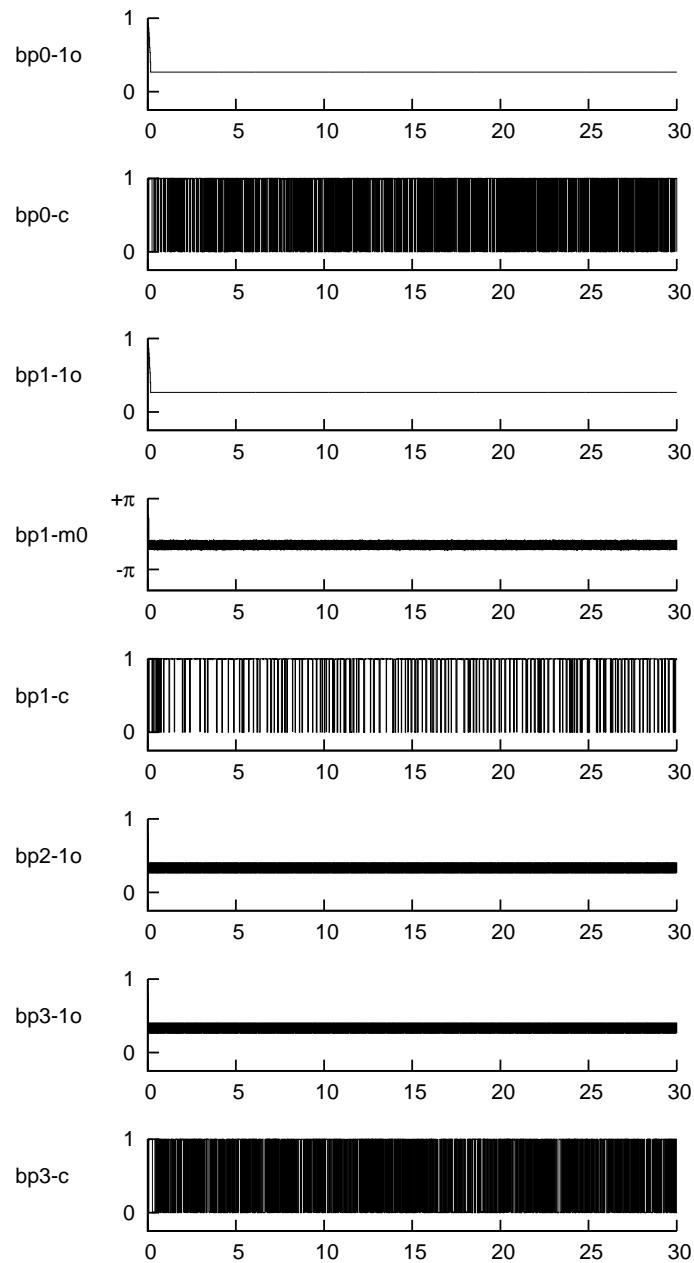
11.7.3 Example quantised controller (Beer's CTRNN model with 16 quanta states)

The neuron model is now quantised, with 16 different quanta states. The network topology is a “random” network with 2 inputs per node. As described in section 8.6 it is not actually random; the topology has no geometric layout and so is initially randomly connected, but these connections are subject to evolution and are inherited by children. The number of inputs per node is also misleading — although the k value is enforced within the neural network, it is not enforced for connections to sensors and motors, or for inter-cylinder connections, so there may well be neurons with more than 2 inputs. There are 4 neurons per network and they are updated synchronously. This particular network achieved a score of 0.728 in the final population.



This graph shows the networks of two connected cylinders. The cylinder on the left has a ball joint connecting it to some other (unplotted) cylinder. On two of the axes the motor for this ball joint receives a constant value. This is not so interesting, as these signals show only that a point attractor exists. The network on the right is from a child cylinder of the network on the left, they are connected with a universal joint, but the motor on both axes only shows constant values, which again is not so interesting. The

only motor in this network which is driven with an oscillating waveform is “bp1-m0”, which is the node to the top-right of the left cylinder. The neuron driving this motor is bp2-1, plotted to the bottom of the right cylinder. This neuron has no inputs, so its internal dynamics must be wholly responsible for the oscillating motor signal.

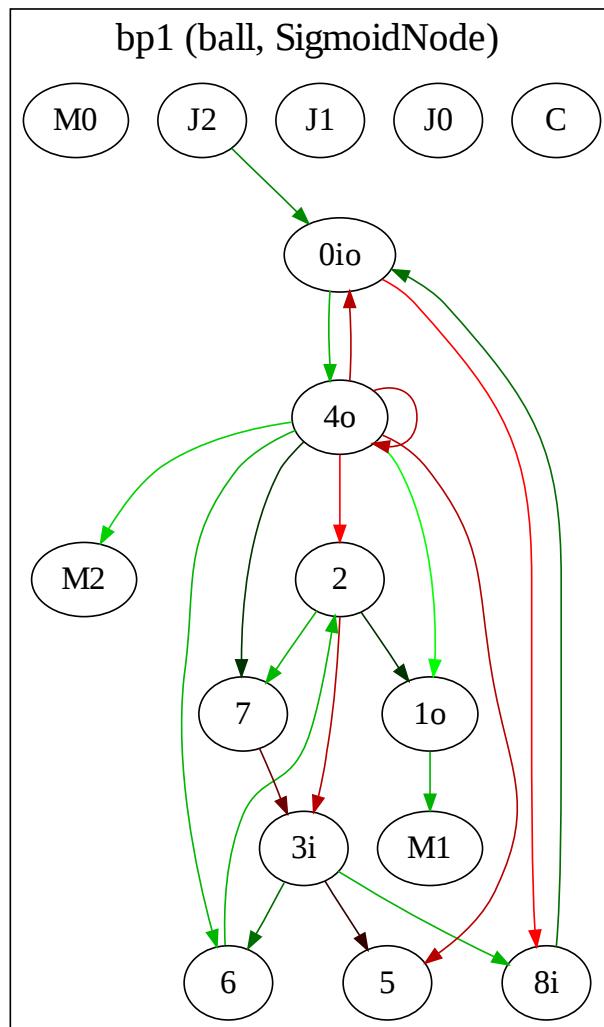


This signal plot shows all the output of all the neurons in the robot’s four cylinders.

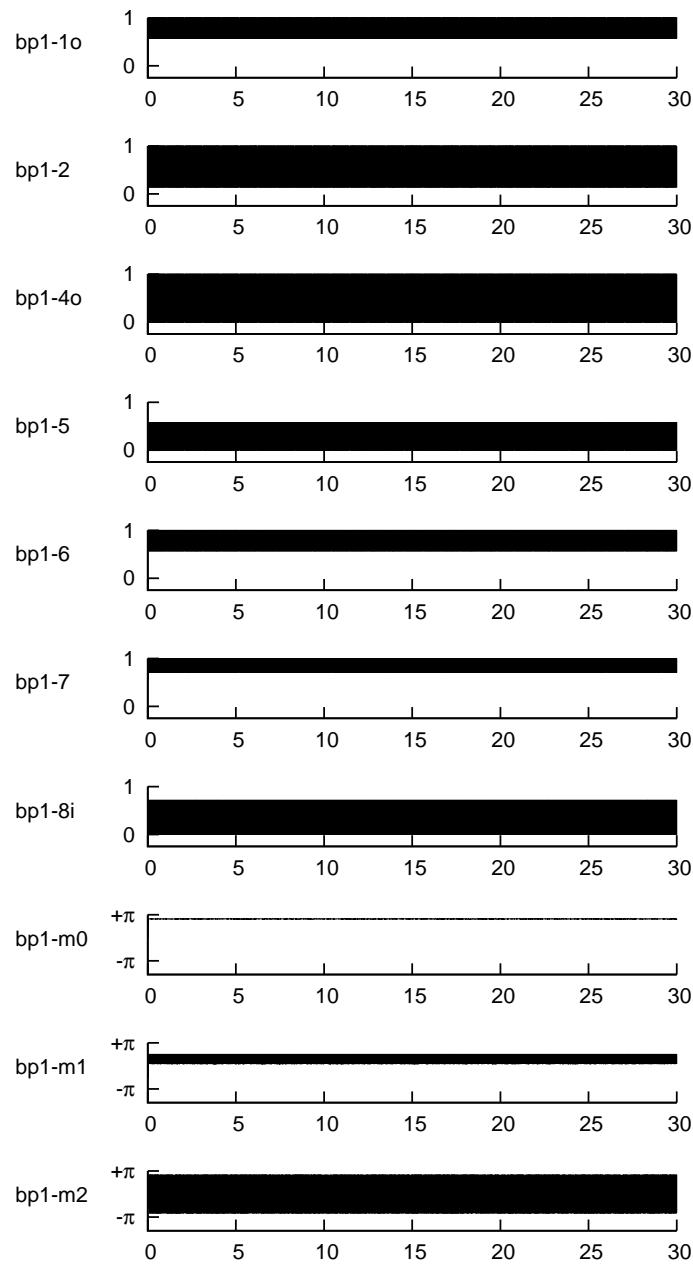
der nervous system that had non-constant signals. There is only one motor which is driven with a non-constant value — “bp1-m0”, which is driven by a Beer model neuron labelled “bp2-1o” (the “o” suffix means it can be used as an output to drive a neighbouring cylinder, as in this case). This neuron had no input connections so its dynamics were wholly internally generated. The signal plot shows that the output of this neuron is oscillating very quickly within a small amplitude range — so quick, in fact, that the output appears at this resolution to be a thick black line.

11.7.4 Example quantised controller (sigmoid model with 8 quanta states)

This controller has 9 sigmoid neurons per cylinder, with random topology and two inputs per neuron (excluding sensors and inter-cylinder edges), using a quantised model with 8 quanta states and synchronous updating. This particular network achieved a score of 2.975 in the final population.



This network plot shows the network of a single cylinder. It has a ball joint with its parent driven by two motors (the third is not connected, so will be unused). Both motors (M1 and M2) are driven by standard neurons (4 and 6 respectively).

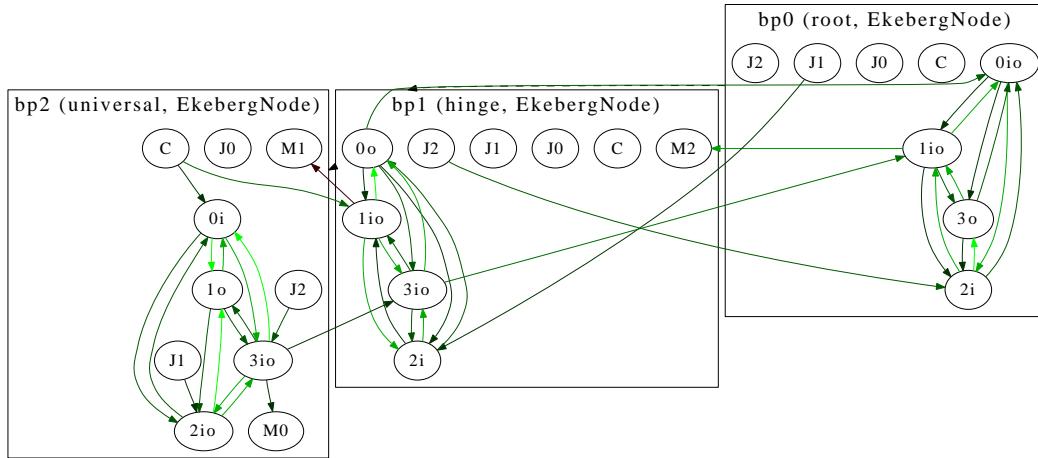


In this signal plot several neurons are oscillating very rapidly. At this resolution the rapid oscillation appears as a thick black line. The oscillations are repeated throughout

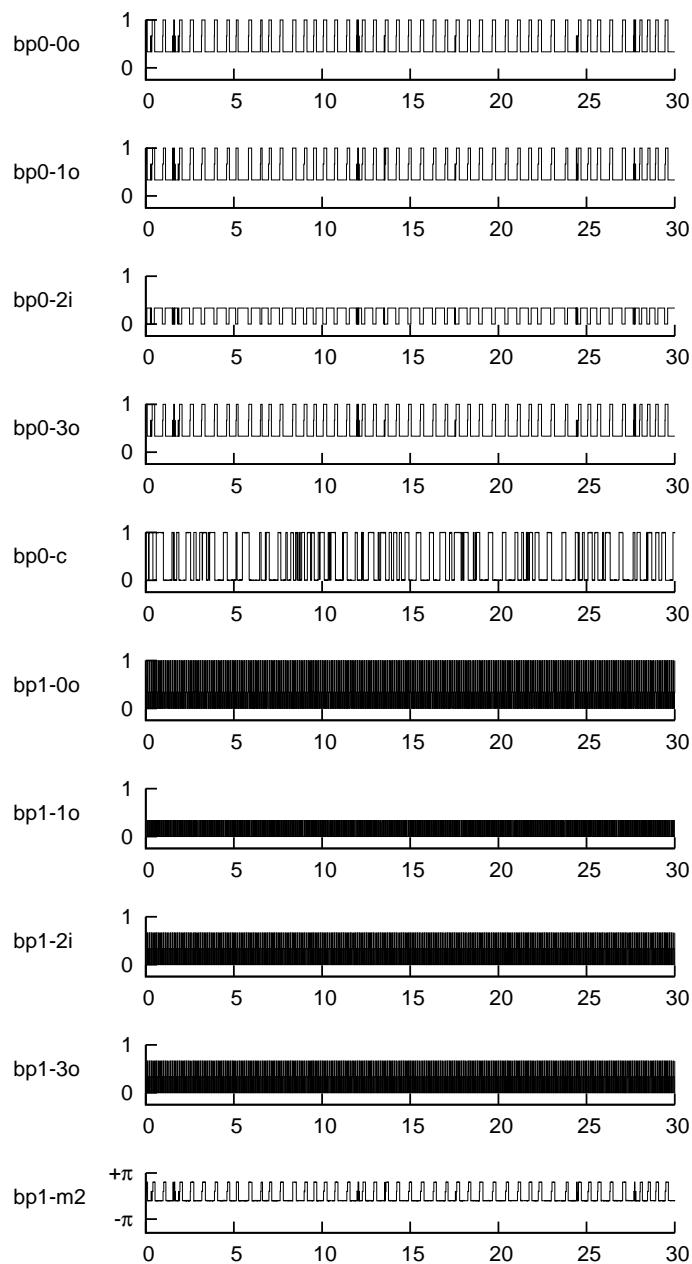
the whole 30 seconds, and so are stable attractors. The two oscillating motor outputs can be seen at the bottom of the plot.

11.7.5 Example quantised controller (Ekeberg model with 4 quanta states)

The neuron model is Ekeberg and is quantised with four quanta states. The topology within each cylinder network is fully connected and updated synchronously. This particular network achieved a score of 1.939 in the final population.



As this robot consists of only three cylinders with a few neurons it has been practical to plot the whole nervous system here. Visually there are a large number of green connections, due to only positive connections weights being used in Ekeberg networks. The centre cylinder has a hinge joint with the root cylinder (to the right). The hinge axis motor (M2) is driven by an output neuron from the root cylinder.

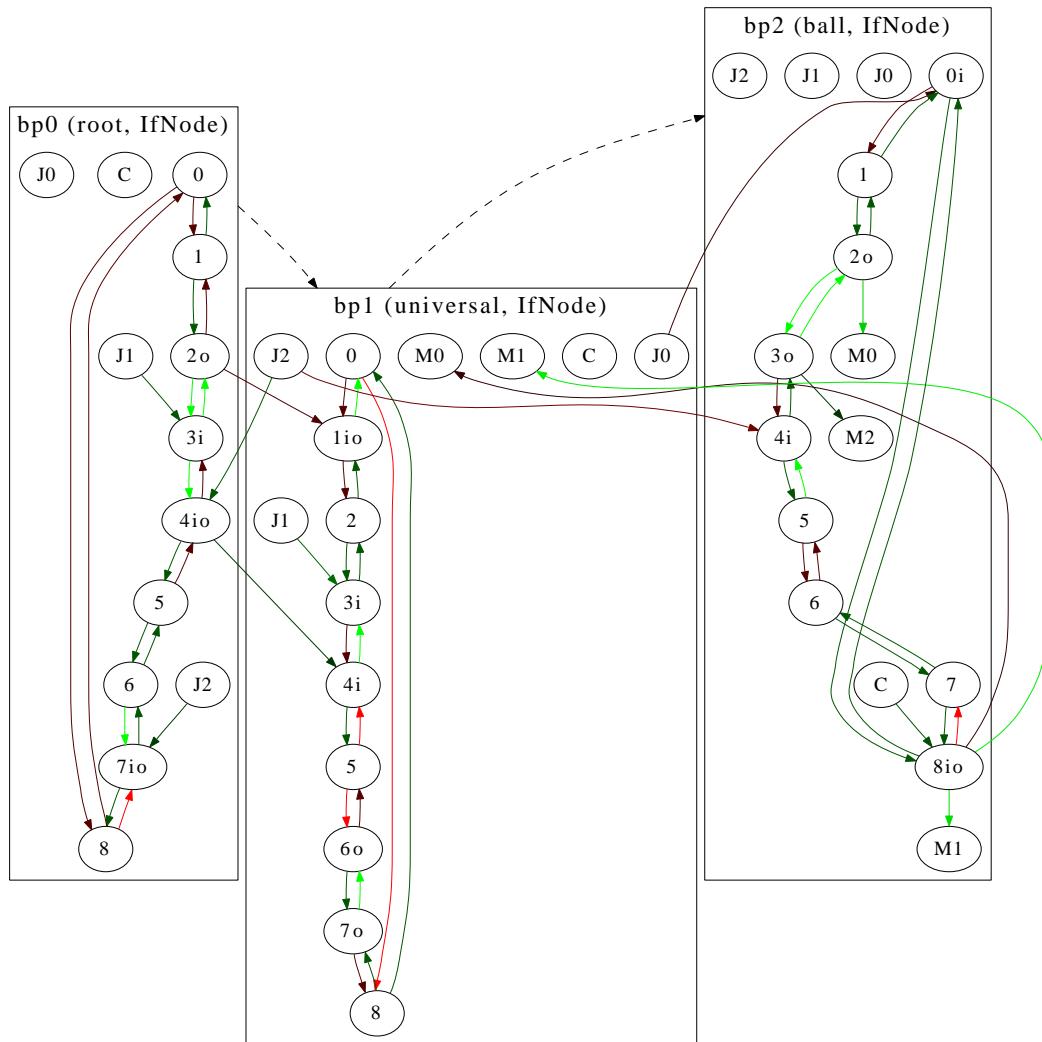


This signal plot shows activity from neural networks of the first and second cylinders. As can be seen, the network dynamics generate regular, cyclic patterns. The signal trace from the hinge joint pointed out in the previous paragraph is plotted on the bottom row (“bp1-m2”), showing a stable oscillating (but not identical) waveform created by the dynamics of the network. This motor signal is driven by the neuron bp0-1,

which is an output neuron from the root cylinder. The other neurons in the root cylinder network are oscillating at a similar frequency, suggesting a dynamic attractor has been formed. The suggestion that the contact sensor may be oscillating and driving the dynamics can be ruled out, as the contact sensor in the root cylinder is disconnected.

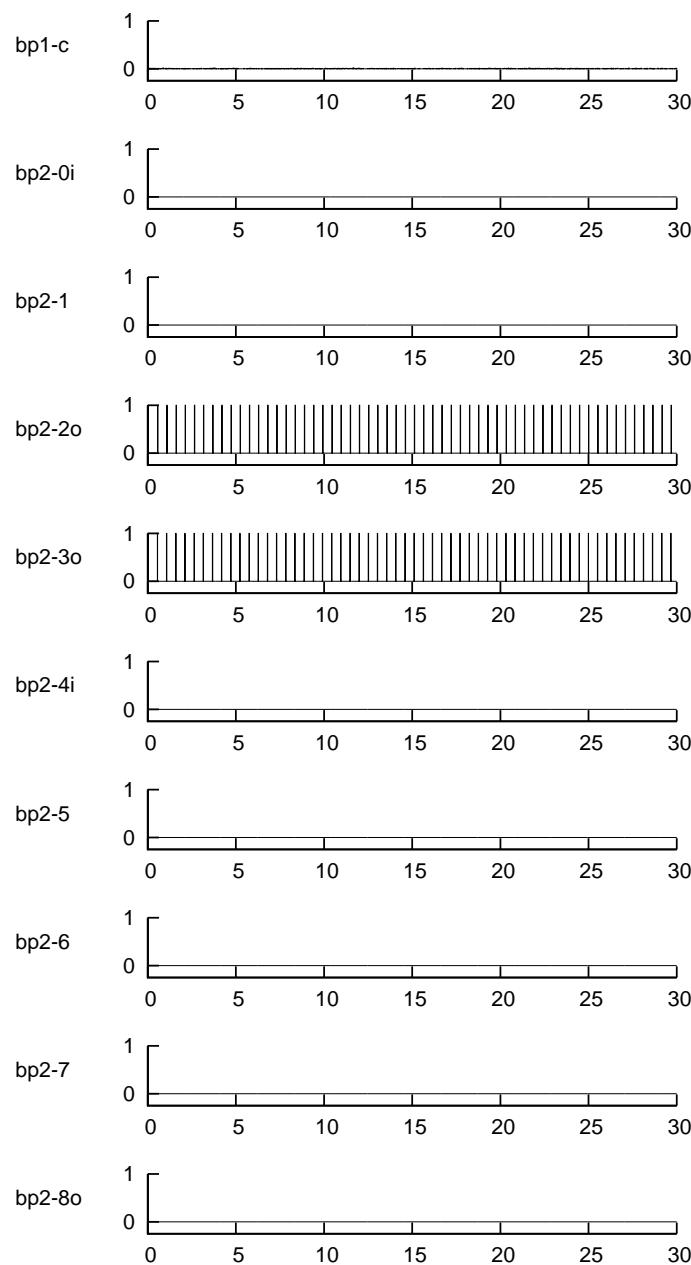
11.7.6 Example quantised controller (integrate-and-fire model with 4 quanta states)

This is a three-cylinder robot with a 9 neuron network per cylinder. The neuron type is a 4-quanta version of the integrate-and-fire-model with synchronous updating. This particular network achieved a score of 1.093 in the final population.



Since this is a relatively compact robot the whole nervous system has been plotted. The network topology was initially one dimensional with neighbourhood size 3, and the cyclic, bidirectional nature of this topology is apparent in the plot. The mutation

operators have altered the initial topology slightly by deleting some nodes. The cylinder that will be examined here is the central one. It has a universal joint with the root cylinder.



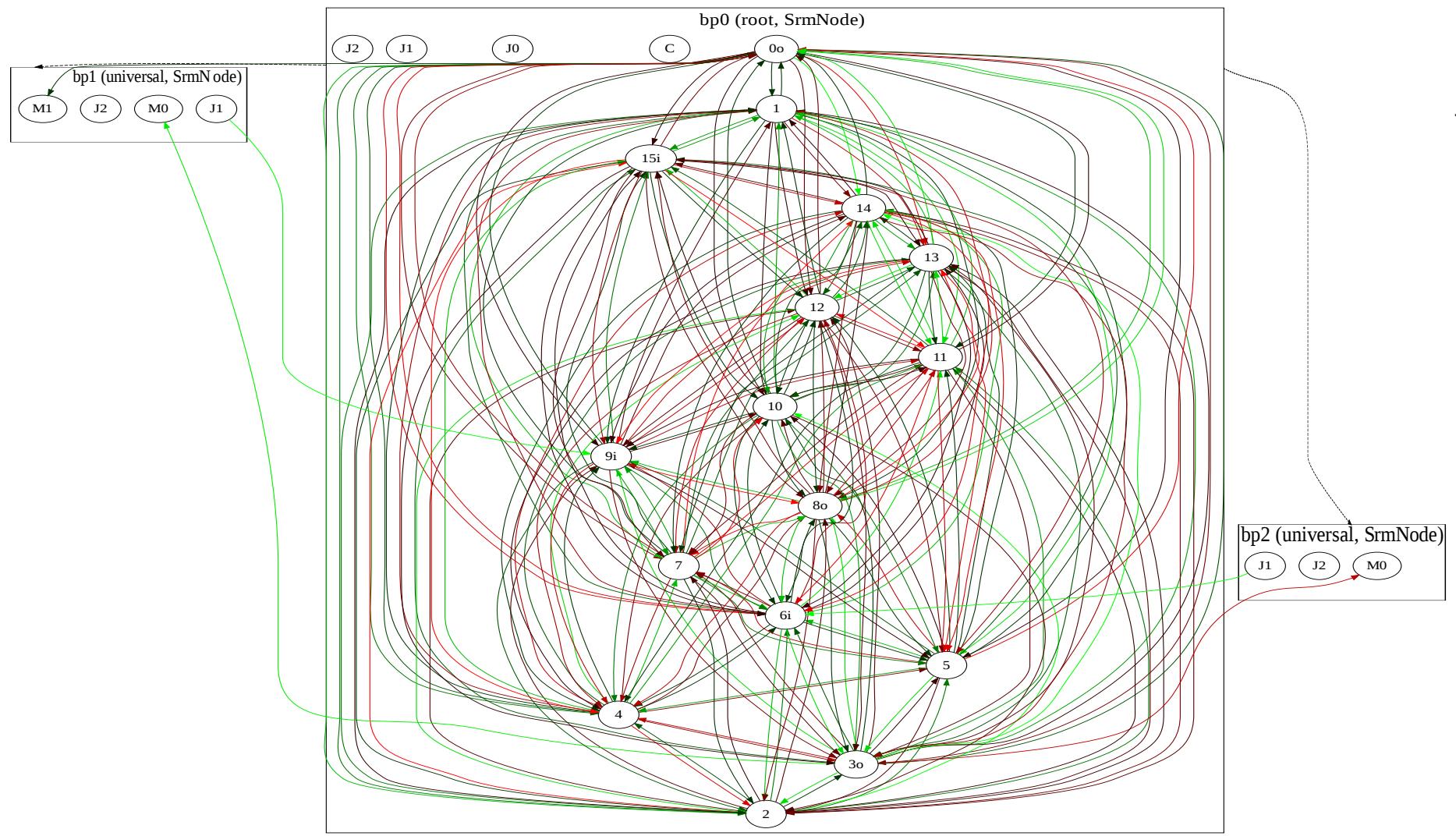
This is a signal plot of a spiking network. Unlike the previous graphs, the recorded signals here will be 0 unless the neuron happens to be generating a spike at that time.

The initial state is still random, which accounts for the unusual appearance of so many signals that are quiescent — the initially random internal state is high enough to cause the neuron to fire in the first time step of the simulator, but they never fire beyond that. Here we have two neurons that are exhibiting stable, oscillating dynamics (bp2-2, bp2-3) which together drive the two axes of the motor for this cylinder.

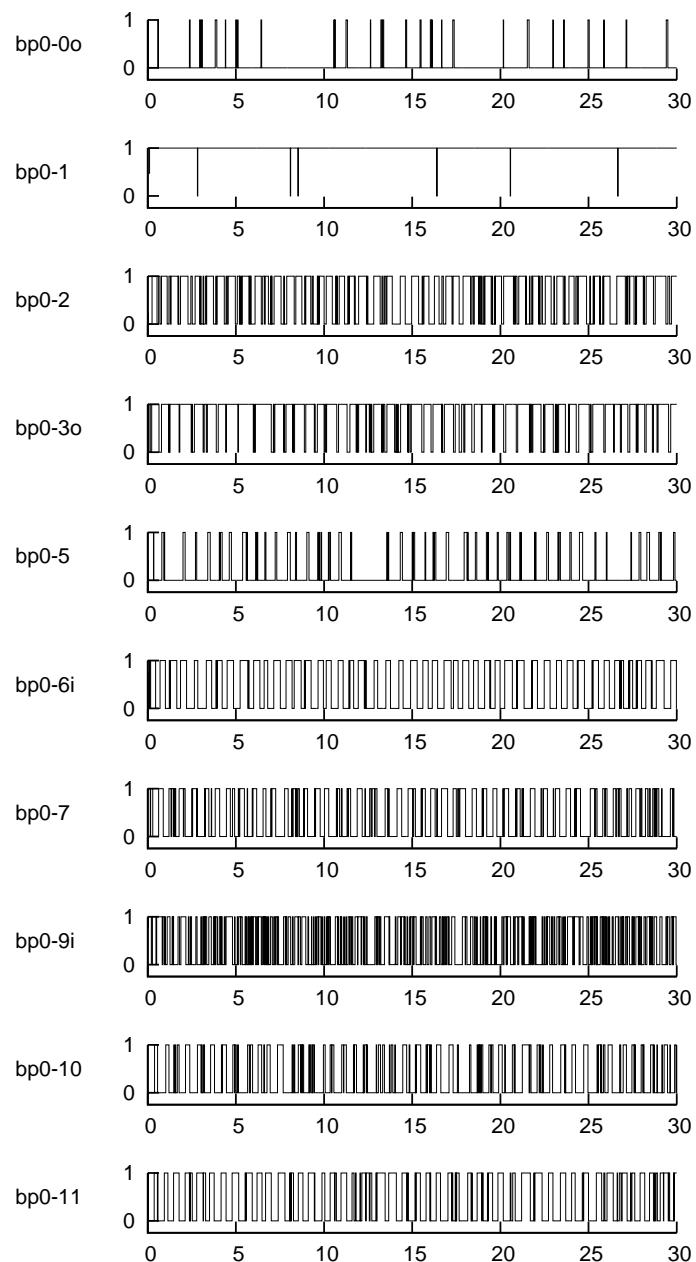
11.7.7 Example quantised controller (SRM model with 16 quanta states)

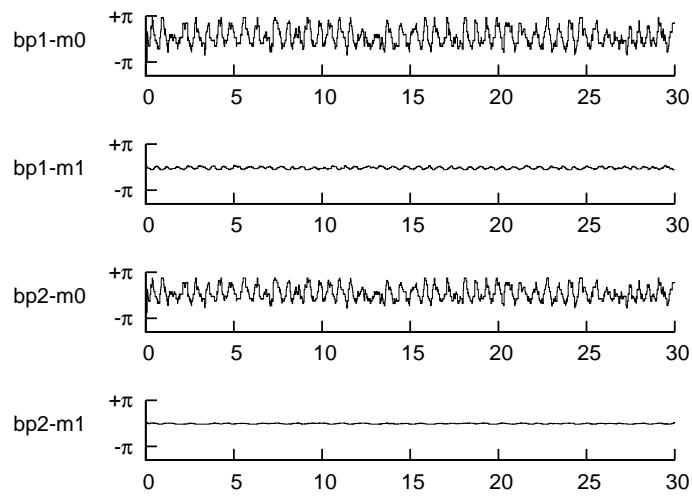
This three-cylinder robot has SRM spiking neural networks. Each network consists of 16 neurons which are fully connected and are updated asynchronously. This is complex and large, so for brevity only one cylinder will be presented. This cylinder is the root cylinder. It has two child cylinders. It has a node (number 3) which acts as an output connection to directly drive a motor in both of the child cylinders (motor identifiers bp1-m0 and bp2-m0). This particular network achieved a score of 2.818 in the final population.

11.7. Example evolved control



This is a complex plot because of the large and well-connected structure of the nervous system. Connections are a mixture of inhibitory and excitatory, and there is no overall discernible structure. The plot shows the root cylinder connected to two child cylinders. The internal nervous systems of the two child cylinders are not plotted here in order to save space and enhance readability. Only the internal motors that are driven by the neural network of the root cylinder are plotted. The root cylinder has no joint, and so no motors, but it directly drives several of the motors in its child cylinders. The root cylinder nervous system plot shows that the joint angle sensors are connected, but they are not since there is no joint (they just return a constant 0 signal). The network has some internal dynamics which generate repeated, rhythmic behaviour. The neuron labelled “3o” (number 3 and an output to neighbouring cylinders) is connected to a motor in each of the child cylinders, and the neuron labelled “0o” is connected to a single motor in bp1.





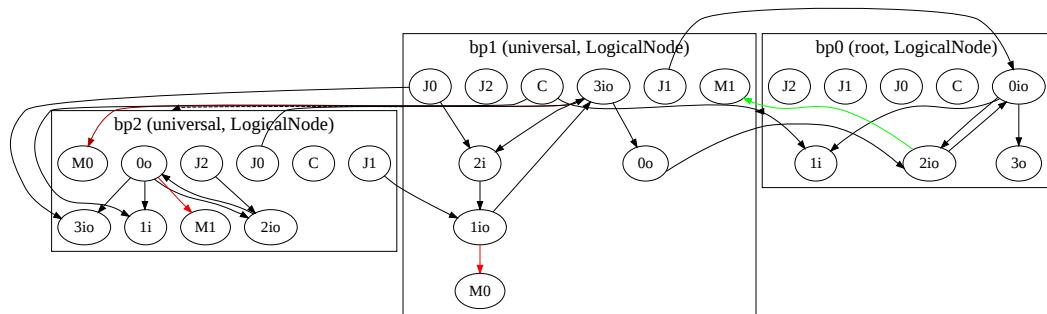
The network generates regular repeating patterns of activity. There is something unusual about this activity graph, which is that the activity is sometimes plotted as what appears to be a constant 1 value, despite the neuron model being a spiking one. This effect can clearly be seen for the plotted “bp0-1” signal. This is caused by the neuron firing repeatedly, on every cycle. This is possible because the refractory period

of the neuron is very small - the SRM neuron specifies refraction as being relative rather than absolute (see the η function, p.55). The specified period of 20ms is the same as our simulated step size, so for the refractory function to have any effect it had to be increased. In the implemented model this was done by stretching it to 200ms. This was an arbitrary number, representing 10 time steps. Even so, it appears that it is easy for a genetic algorithm to construct a network where this relative refraction is overwhelmed by input spikes, generating a persistently spiking signal. The fact that the signal appears as a constant value of 1 is an artifact of the neuron spiking on every simulated time step — the output value from the neuron is always 1, but of course in real life or with a smaller resolution it would reset to 0, and then increase again.

It can easily be seen from the “bp1-m0” and “bp2-m0” signals sent to the motors that the neuron generating these signals is displaying a rhythmic, repeated and stable attractor pattern. Interestingly, the repeated wave pattern does not consist of identical peaks and dips, suggesting some kind of chaotic attractor may be governing the neuron’s behaviour.

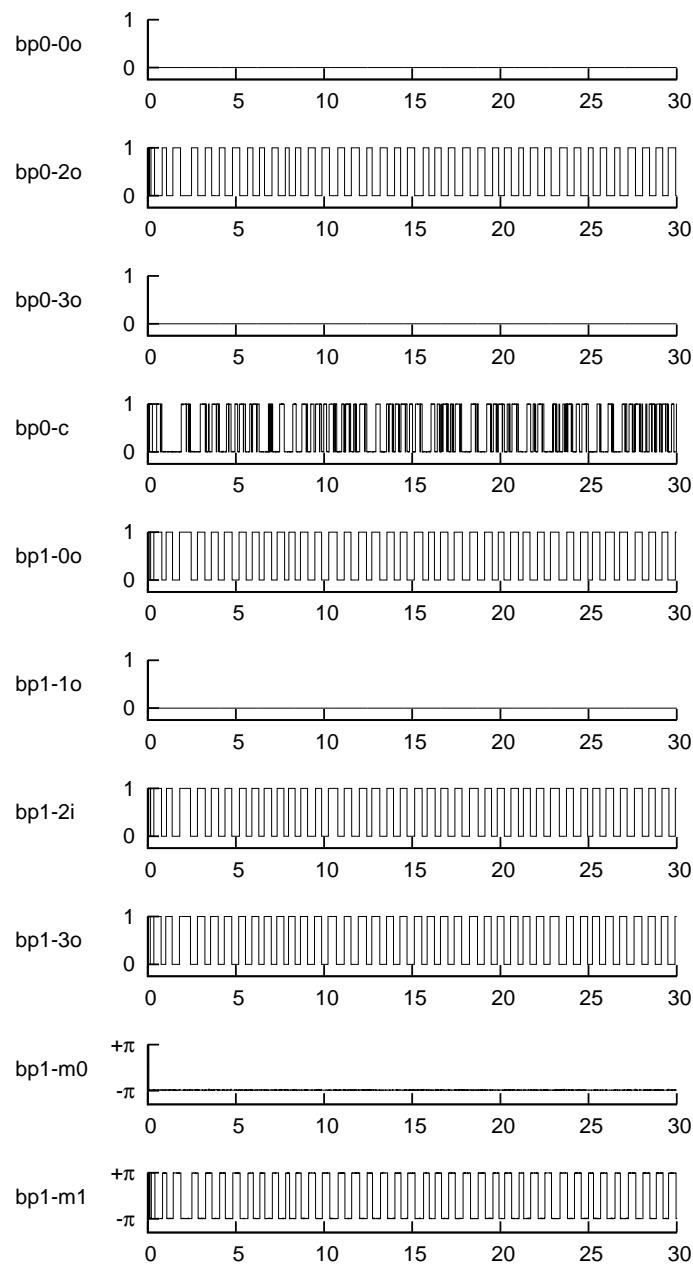
11.7.8 Example quantised controller (Logical model with 2 quanta states)

This robot is made up of three cylinders, each having an associated neural network of four logical neurons quantised with 2 quanta states, connected into a “random” topology with $k = 1$, and updated asynchronously. This particular network achieved a score of 1.906 in the final population.



This is a relatively small nervous system, with few neurons and low connectivity. Most neurons have two inputs (note that sensors and motors are not neurons). The neural network we will examine is that of the root cylinder (to the right), which has no motor (since it is the root cylinder) but does output a signal to the M1 motor of one of its child cylinders (“bp1-m1”). This connection is coloured green as it happens to

be excitatory. Neural connections between logical nodes are unweighted, but to help the evolutionary process we allow edges to motors to be weighted (this is true for all neuron models).

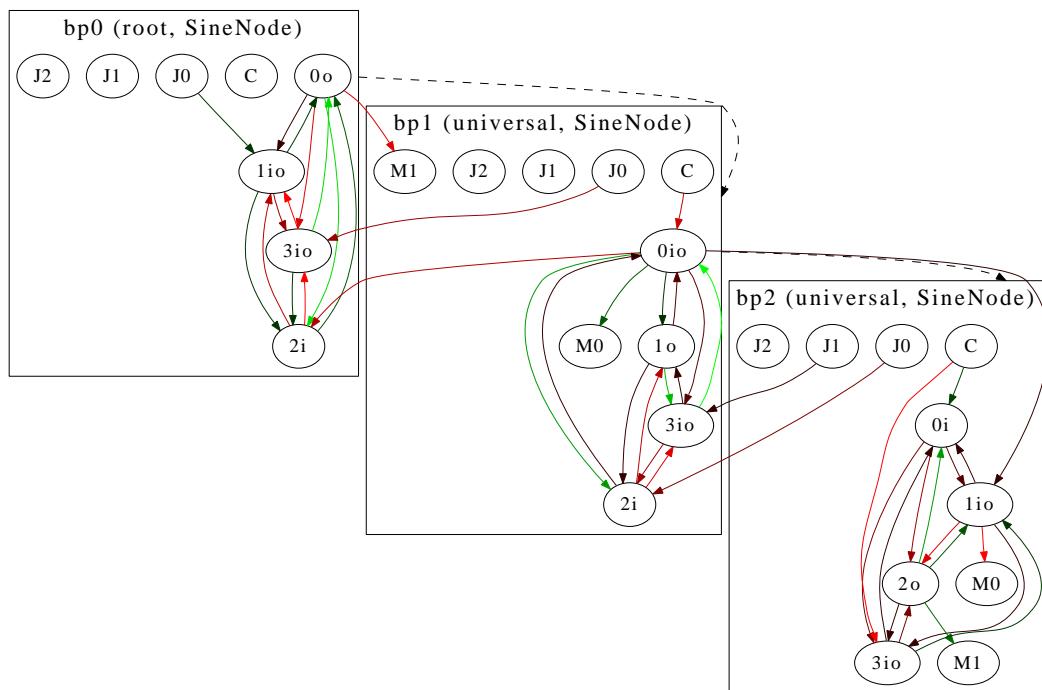


It is clear that the dynamics of the network are generating regular, repeated oscillating waveforms from many nodes. The motor “bp1-m1” is oscillating. This motor is

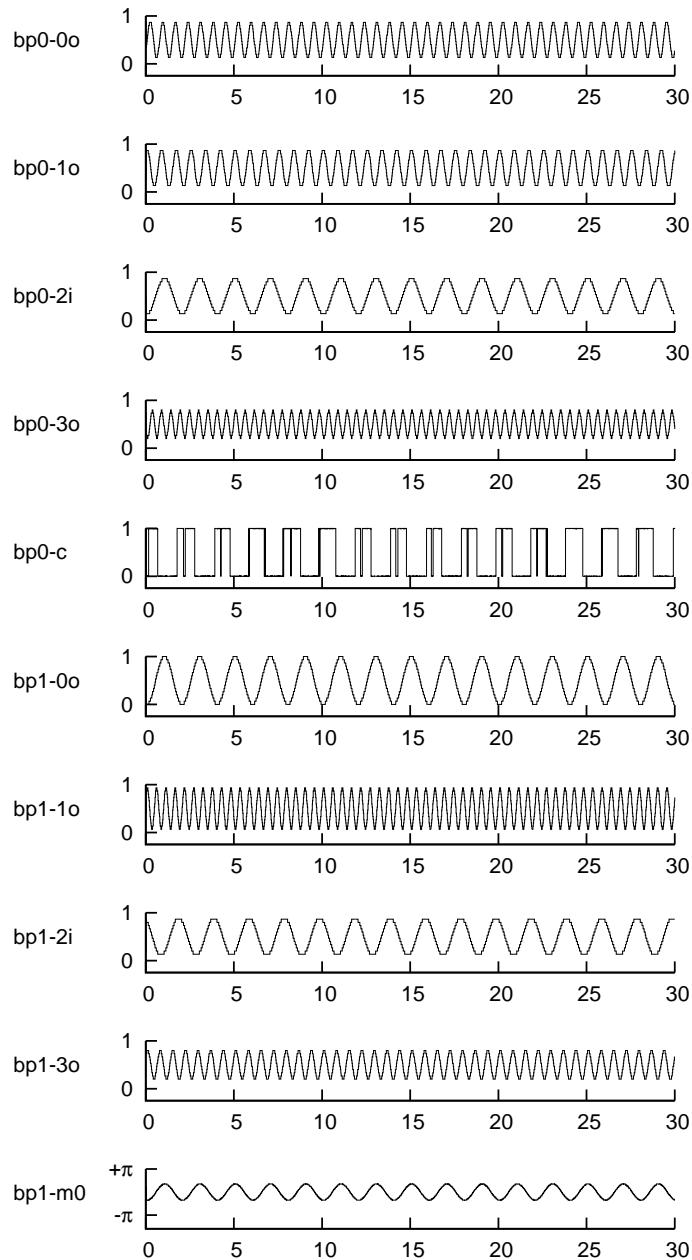
driven directly by neuron “bp0-2”, which is also obviously oscillating.

11.7.9 Example quantised controller (Sine model with 16 quanta states)

This is a three cylinder robot with four sine neurons per cylinder network. The neuron model is quantised with 16 quanta states. Networks are updating synchronously. This particular network achieved a score of 4.474 in the final population.



This is the complete nervous system of the robot. It has three cylinders. Note that there are very few connections. Each motor has precisely one sine neuron to drive each axis of the joint. In this case, we have two children of the root node, with universal joints, and so four motor axes to drive.



The signal plot shows repeated sine wave patterns on every neuron (the square wave is from a ground contact sensor). The quantisation into 16 quanta states is evident, but does not greatly affect the overall shape of the waveforms. The output signal from one of the motors (“bp1-m0”) is plotted on the bottom row. It is a quite reasonable approximation of a sine wave. The signal driving the motor is from the “bp0-0”

neuron which is rendered on the top row. There is not much to say about the dynamics of sine neurons — by definition they generate repeating cyclic patterns of a predictable waveform (in fact, the waveforms will be identical for each cycle, which distinguishes them from those waveforms generated by the previously presented neural networks, where cycles are driven by dynamic attractors and are often visibly non-identical).

One of the reasons that this particular set of signals was chosen is that using 16 quanta states gives a good visual indication as to the effects of quantisation on the sine wave neuron. It should be noted that 2 quanta state sine neurons (i.e. binary oscillators) also worked well to drive locomotion, and are even smaller and simpler to implement.

11.8 Example of evolution — from biped walking onwards

As far as the author is aware, this is the first reported evolution from scratch of a complete morphology and control system for a stable walking biped. This evolution was manually observed and tracked during the experiments described earlier in this chapter. The controller was a sine wave generator network. Surprisingly, the fitness function was, as in all of the experiments described here, the mean velocity along the x -axis, meaning that there was no explicit evolutionary incentive towards stable biped walking. As will be shown, this led the robot along an evolutionary path which eventually threw away biped motion in favour of a snake-like creature which drove its motion by springing forward.

The evolutionary progression described here shows creatures from different generations of the same evolving population. Each creature had the highest fitness score of its generation. The creatures are related — by the time this interesting evolutionary path was noticed the biped walker design was already dominant, taking up all of the “elite” slots that are preserved unaltered and used as parents to seed the next generation. The genotypes of the subsequent generations presented here are descended from the biped walker design. The only reproduction operator in use was mutation.

The evolutionary progression of the walker design is presented here in stages. One or more generations of evolution occurred between each stage. Each stage was manually defined by visual inspection of the best creature from each subsequent generation of the evolving population. If the creature displayed a markedly different morphology or means of locomotion then a video of it was created as a permanent record. Unfor-

tunately there was no facility within the software to permanently store a genotype, so the exact genetic makeup of each creature was not recorded.

11.8.1 A note on lack of reproducibility

Videos were recorded of a creature at each stage of this evolutionary process, providing some visual evidence that this really did occur. Unfortunately, the genotype details of the evolution leading to this biped motion were not recorded, as there were a great number of experimental runs and storing them required disk space. This run was observed purely by chance. It was not intended to manually check all evolutionary runs for interesting creatures as it would have been prohibitively time consuming and was not required by the experimental design. It is not known whether similar creatures evolved during other evolutionary replicates. It is standard practice in genetic algorithm research to not permanently store the individual genotypes that make up every generation of every population, partly for reasons of conserving disk space, and partly because the experimental design does not call for it. Nevertheless, the evolution of a successful walking bipedal design was very surprising given that this was not an aim of the research, and is important enough to document in this thesis.

The software written for this thesis was originally designed to store every single genotype of every generation permanently. Unfortunately this required a lot of disk space: around 1GB for each replicate of the experimental setup described in section 11.4. With 533 replicates this would amount to approximately 533GB, which is actually quite feasible – if it were desired to conduct experiments that required post-analysis of whole evolutionary runs it could be done with current technology. However, permanent storage of generations was disabled during these experiments as the quota allocated on the computational cluster was only 1GB (as a comparison point, with permanent storage of generations disabled, the dataset for each experimental replicate takes just under 10MB on average, allowing a batch of 100 replicates to be processed in parallel). More importantly, there was not thought to be a reason to request an increase in quota to store all data for post-analysis, as the experimental design did not call for it.

It should be noted that the experimental replicates might be exactly repeatable given the initial random seed and setup conditions. Unfortunately, as already mentioned earlier in the thesis, the program code makes use of some non-deterministic operators, an issue that could be fixed with some effort to identify and replace them

all. There are other parameters which would require consideration for complete reproducibility: the possibility of (very) small rounding errors on floating-point arithmetic from CPUs of different models or manufacturers, and obviously the versions of all libraries that affect the results of both evolution and simulation (the “Open Dynamics Engine” being the prime example) would need to remain constant, or be checked very carefully to ensure that updates did not change experimental results. Maintaining exact and complete reproducibility of full evolutionary runs is a desirable, but difficult task.

11.8.2 The observed evolution

The evolution has been divided into five distinct stages as described in the previous section:

Stage 0 The creature displays full stable biped walking. It has distinct identifiable body parts, consisting of feet, legs, a trunk or neck, and a head. The long head weights the body, causing it to turn in whichever direction the head moves towards.

Stage 1 Biped walking has been abandoned, and the head is much smaller and in almost constant contact with the ground. Motion is no longer driven by the feet, but by the knees, allowing the creature to deliver alternating force in the required direction, with the head coming into contact with the ground to prevent the robot from falling over.

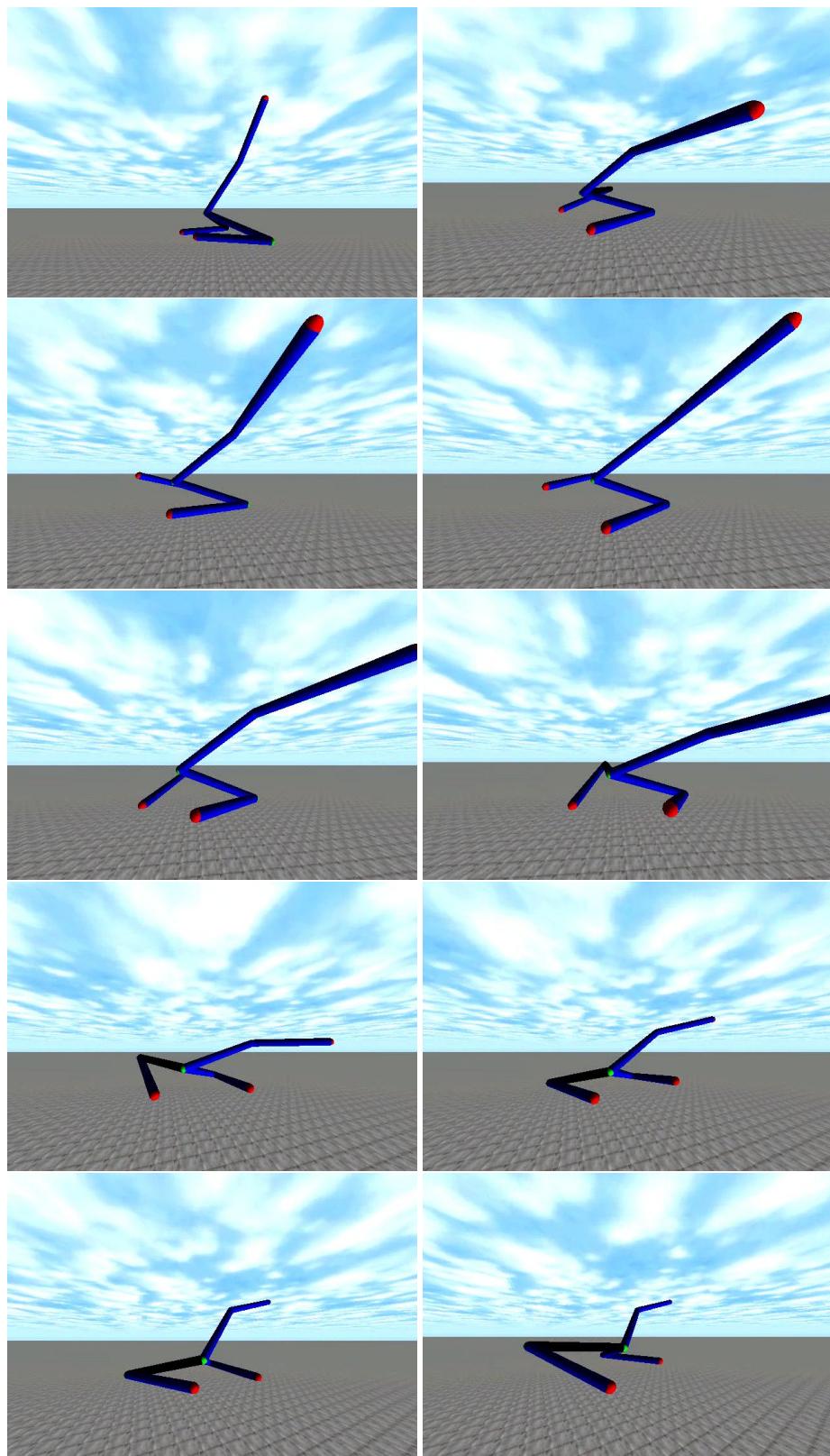
Stage 2 The creature loses a leg, however it can still drive itself forward, and there seems to be less contact with the ground, and hence less friction, as the head is no longer being driven into it due to instabilities of the alternating leg motion.

Stage 3 The lower leg has become slightly longer, and is used to deliver more force, thumping into the ground behind the creature to drive it forward.

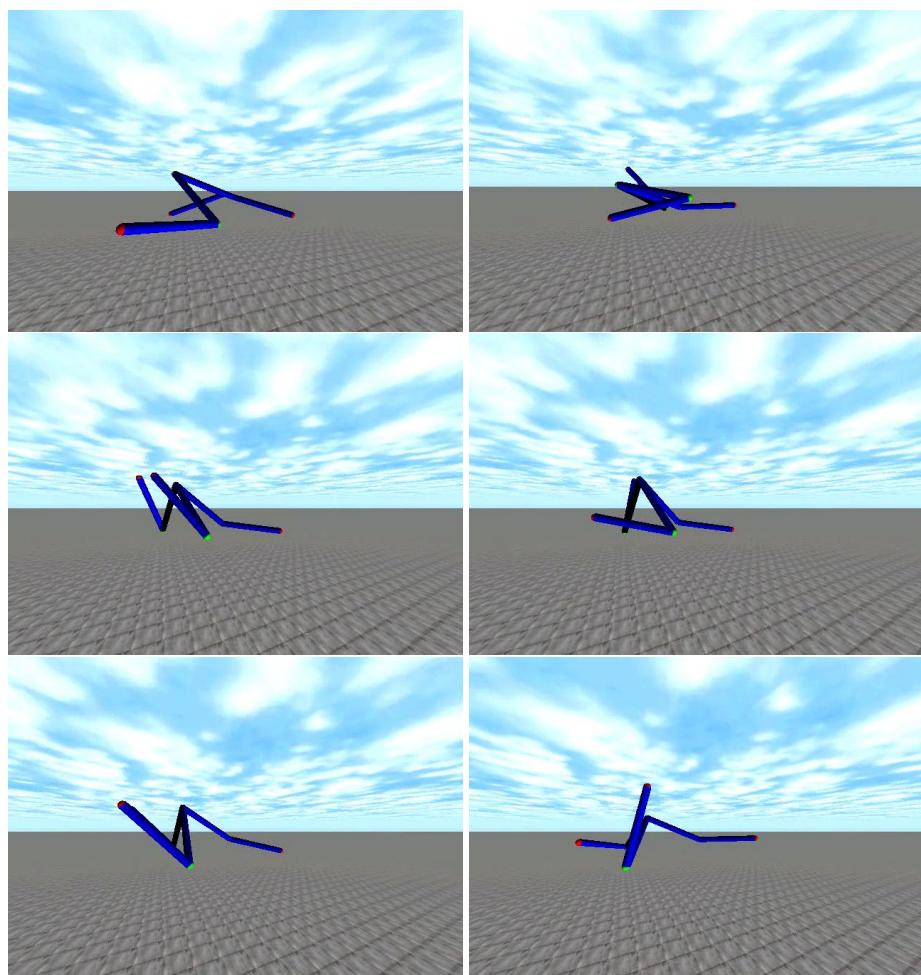
Stage 4 The creature displays better global coordination, with a distinct periodic rippling motion along its body creating forward momentum. Its rear body parts that were formerly parts of an identifiable leg have become longer, giving it better balance and the ability to raise and flatten its centre without falling over.

The locomoting behaviour of each stage is presented here as a sequence of still images, reading from left to right.

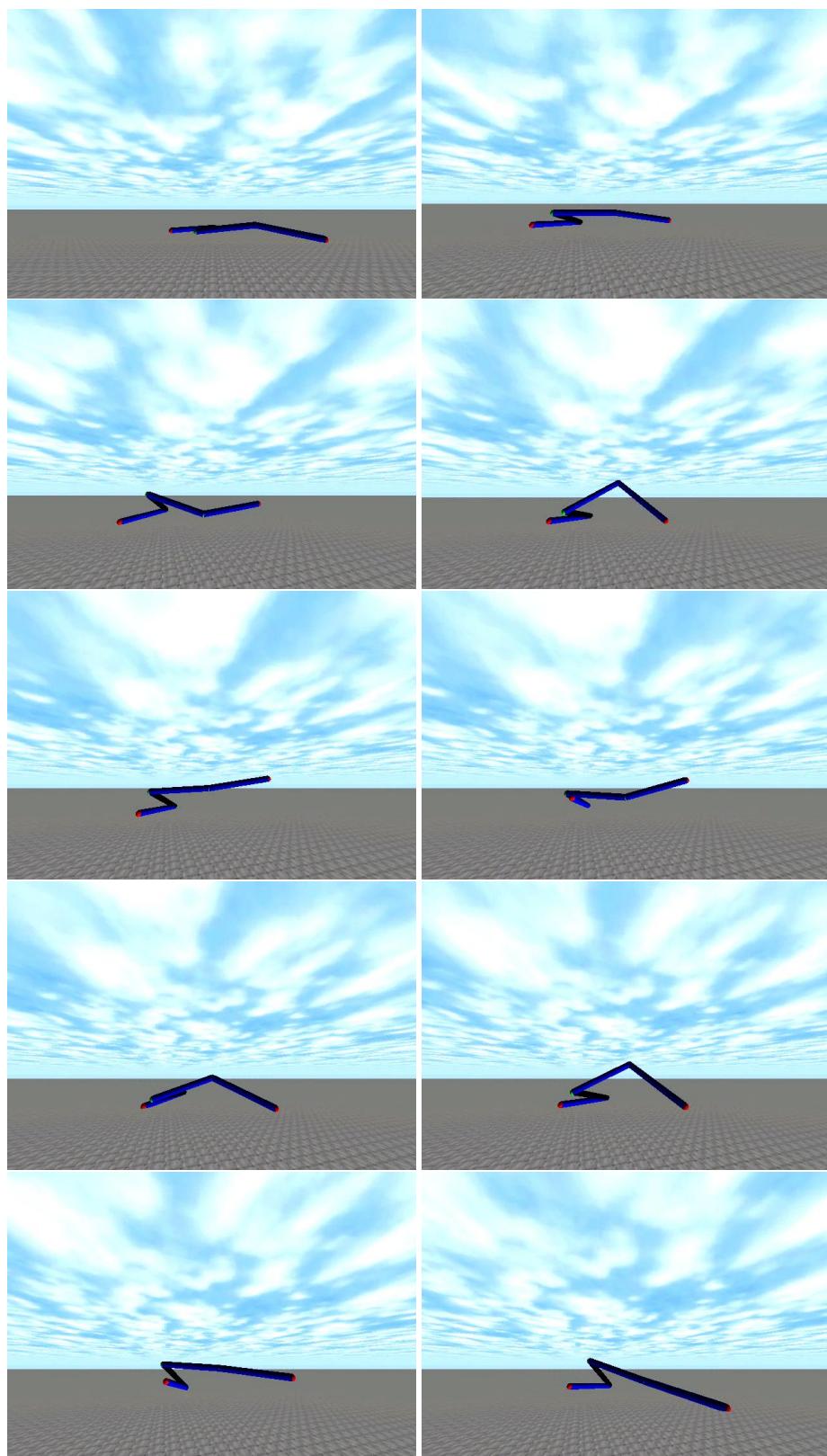
11.8.3 Stage 0 — walking biped



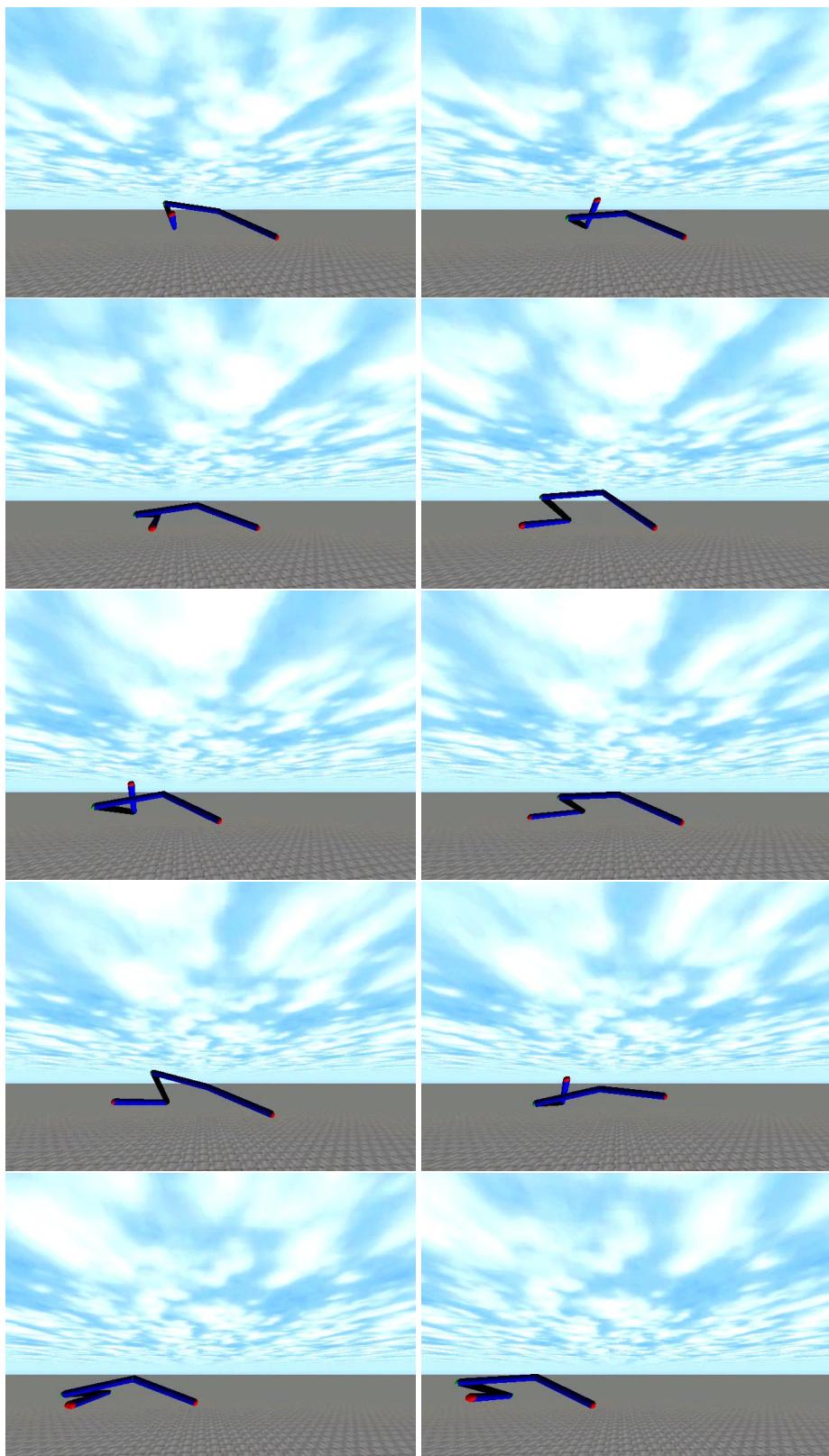
11.8.4 Stage 1 — smaller head, pushing forward on knees



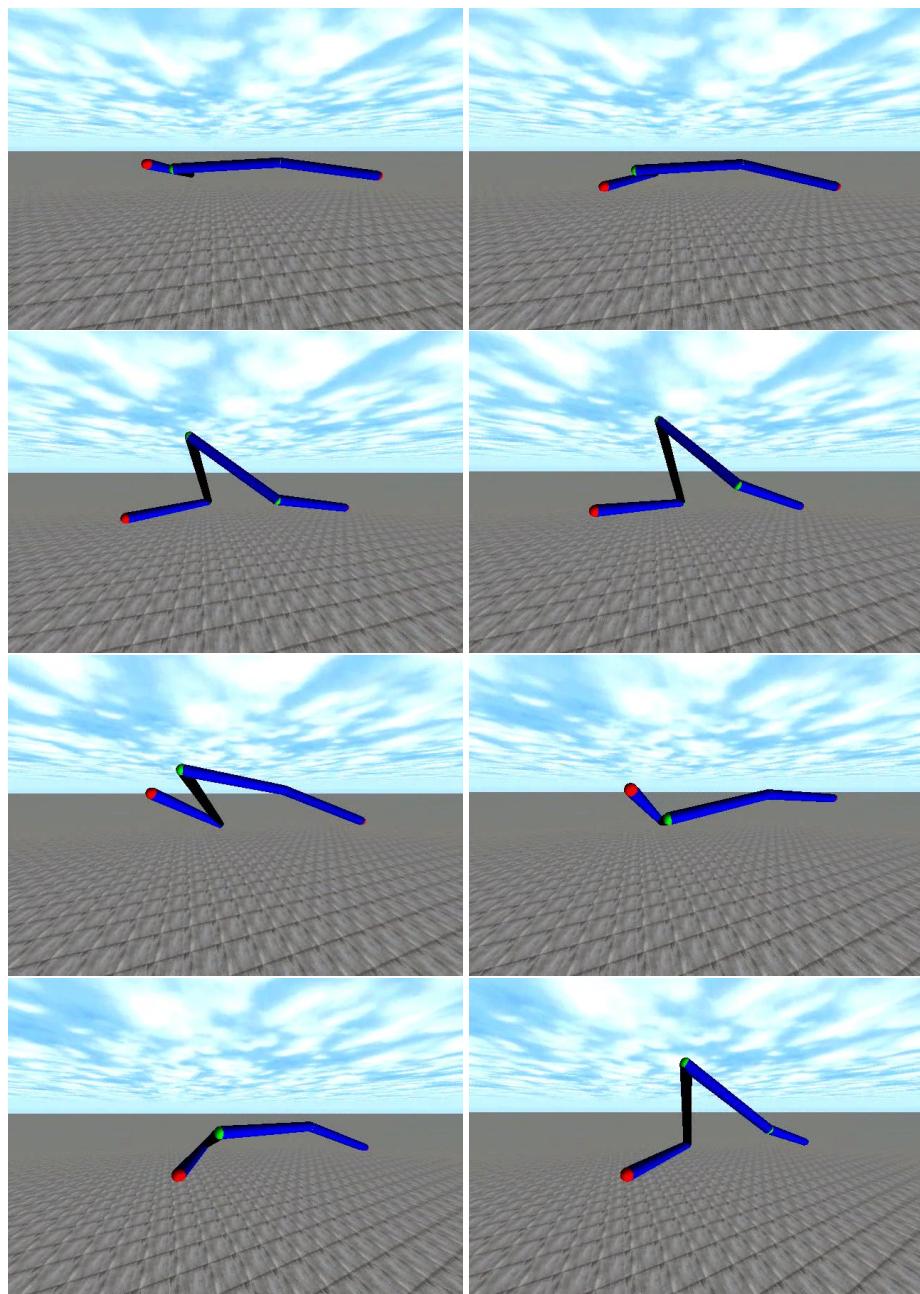
11.8.5 Stage 2 — loses a leg



11.8.6 Stage 3 — stronger leg



11.8.7 Stage 4 — better global coordination, longer leg



11.9 Summary

The co-evolution of genomes which combine creature morphology and neural control systems is an active field of research. In the last 15 years, increases in computational power have made the prospect of simulating and evolving three-dimensional virtual creatures a reality. Unconstrained evolution from scratch has been used many times to evolve both the bodies and neural network based nervous systems of creatures that display locomoting behaviour [34, 38, 204–206, 240, 242, 243, 262, 295, 296, 340, 348, 390–393, 397, 398, 435, 465]. Without exception, these works have all used neural network models with floating-point arithmetic. Various neuron models have been implemented, with the most popular being parameterised waveform generators (sine, sawtooth, or square wave) and sigmoid neurons. None of these works has questioned whether it is necessary to use neural models which rely on floating-point arithmetic, or whether models with reduced precision may suffice. There is a big motivation to explore this question, as quantised models do not require floating-point arithmetic units, and so are simpler to implement and consume less power — a property which is particularly important for mobile robotics applications.

This chapter has explored whether quantised neural networks can be co-evolved with morphology to generate complete locomoting virtual creatures, and if so, how performance degrades when quantised models are used and arithmetic precision is reduced. It was unknown whether quantised networks would work at all — it was possible that lower precision arithmetic would alter the dynamics of a neuron model, making it unlikely to generate the kind of behaviours that underlie locomotion control.

A fractional factorial experimental design was created to test the hypothesis that quantised neural models could be evolved to drive locomoting behaviour, and to quantify how the performance of evolved neural controllers would change in response to lower arithmetic precision. The performance metric used was the velocity along a single axis measured over the complete simulation time. The experimental design varied other parameters that might affect the performance of evolved controllers. Experiments were carried out on a PC-based computational cluster, with 6.6 million physics simulations being carried out in total. ANOVA modelling of the results showed that there were several significant factors: neuron model, a combined factor representing the size of the population and number of generations to evolve, and the type of neural synchronisation. Additionally, there were two interaction factors that were significant: an interaction of the neuron model and the type of neural synchronisation, and an in-

teraction of the neuron model and number of quanta states.

The result of the experiment was that creatures displaying locomoting behaviour driven by quantised neural networks were successfully evolved. The sine model was far superior to all of the rest, with a mean velocity more than 2.75 times greater than the second best performing model. The success of the sine model was surprising given that it ignores sensory input and only drives motors with parametrised sine waves. This indicates that, for the locomoting task, the ability to easily generate cyclic patterns is more important than reacting to sensory data. All of the neural models displayed the ability to generate locomoting behaviour to some degree, with the logical model, sigmoid model, spike response model, and Taga's model all performing reasonably.

The quantised models of the different neuron types were compared to the floating-point models. The results showed that there was no significant difference in locomoting performance between those implementations using floating-point arithmetic and those using reduced precision arithmetic for at least some of the quanta levels. In the case of the best model — the sine model — there was no significant difference in reducing precision to 1-bit (2 quanta). This is an important result, as it shows that simple binary oscillators perform equivalently to floating-point sine wave neurons on this task. Results for the sigmoid model also showed no difference between the performance of floating-point controllers and those using reduced precision, even down to a 2 quanta model.

This pair of results for the sine and sigmoid models are important, as these models are the most widely used in evolutionary robotics research to generate locomoting behaviours. The implication is that, for locomoting applications, evolved floating-point neural controllers can potentially be replaced with evolved quantised neural controllers without any observable degradation in performance. This is particularly important for evolutionary robotics, since quantised networks do not require floating-point arithmetic units, and hence are simpler to implement and consume less power, meaning that autonomous mobile robots with limited energy stores will be able to conserve energy.

The combined factor representing the population size and number of generations was found to be significant, with a population/generations value of 100/100 producing much better controllers than a value of 50/50. This shows that there is a reason to use more computational power beyond that provided by the 50/50 case. The 100/100 case required significant computational resources, representing the upper limit of what was feasible for this research. It is likely that greater computational resources would yield increased performance, though this will obviously produce diminishing returns

at some point.

Timing was found to be a significant factor, and one that interacted with the neuron model. Beer's CTRNN neuron model, the sine model, and Taga's model were all found to have increased performance with a globally synchronous timing scheme. This shows that it is important to have some degree of synchronisation between certain neuron models. However, even with a completely desynchronised timing scheme, locomoting behaviour was achieved with a performance level comparable to synchronous timing for many neuron models. More research is needed to establish the optimal timing regime — it seems likely that some hybrid approach, synchronising pairs or small groups of neurons, may work best. Biological neural networks appear to encode timing and data into a single continuous real-valued signal. More research is needed to establish how neural coding works in living creatures, and whether this would transfer to virtual creatures.

Chapter 12

Conclusions

12.1 Summary

This research has shown that quantised networks can perform equally as well, or even better, than floating-point networks in generating realistic and practical robot controllers for two well known control problems. Quantised networks are simpler, and can be implemented in circuits that are smaller and consume orders of magnitude less power than complex floating-point units.

The pole balancing problem has previously been solved several times by using genetic algorithms to evolve neural network controllers. Without exception, these networks were implemented using floating-point arithmetic. The research presented in this thesis has shown that quantised neural networks, using reduced precision, can successfully be evolved to solve the pole balancing task, and that the performance of these evolved controllers is comparable to that of evolved floating-point controllers.

Likewise, the locomotion problem for evolved virtual creatures has previously been solved several times by using a genetic algorithm to evolve neural network controllers. Again, without exception these neural networks were implemented using floating-point arithmetic. The research presented in this thesis has shown that quantised neural networks can successfully be evolved to control locomoting behaviour in virtual creatures with co-evolved morphology, and that the performance of these evolved controllers is comparable to that of evolved floating-point controllers.

It seems to have been assumed that complexity is necessary in order to achieve many of the features of stable locomotion in evolved creatures with arbitrary morphologies. This research has suggested that complexity is not necessary, and that locomotion in evolved morphologies can be achieved with very simple control systems. In

particular, this research has presented a walking biped that was evolved from scratch by the genetic algorithm. Bipedal walking is recognised as a difficult control task, and has been heavily researched. Controllers for bipedal walking tend to be complex, with large amounts of sensory input being gathered and processed by powerful CPUs in order to calculate the motor forces required to achieve stability. This thesis has presented an evolved bipedal morphology that displayed stable locomoting behaviour using only synchronised sine wave oscillators. This shows that complex locomoting behaviour can be generated by non-complex parts when a genetic algorithm has the freedom to create and optimise both together.

The best locomoting controllers used sine neurons to drive motor control. The sine neuron model generates patterns but completely ignores sensory inputs. It has previously been shown that central pattern generators produce more stable locomotion gaits than reflex based generators [127], so although the ability to produce stable locomotion from only pattern generation was not unexpected, it was surprising that controllers which did so vastly outperformed those that had the potential to use sensory input in a useful way. This suggests that more research is needed to establish optimal sensory schemes for evolved virtual creatures.

The success of the sine neuron model in comparison to others in solving the locomotion problem in evolved creatures has shown that, for this particular task, pattern generation is more important than either reflex based control or modulatory sensory perception. The importance of oscillation is demonstrated by the success of some 2 quanta controllers, which are only capable of producing an oscillating binary output (although it was obviously possible for the neuron to control the timing of transitions in its output signal, which was not possible with the sine model). For many models (sigmoid, Beer's CTRNN, Ekerberg's model, Taga's model, and the sine model) there was no significant difference between the performance of the two quanta controller and the floating-point controller. For two other models (logical and spike response model) the two quanta controller actually outperformed the floating-point model.

The poor performance of the more biologically valid neural models in comparison to the sine neuron can be attributed to their difficulty in generating oscillating patterns. With the sine model, the genetic algorithm must specify frequency, phase offset and amplitude, but it is guaranteed that the output signal will oscillate. With other neuron models a parameter space must be explored to discover subsets of the space where the output will oscillate, and for some models, oscillation can only be achieved by connecting multiple neurons together and specifying certain parameters. This makes

it harder for the genetic algorithm to discover oscillation. The problem of locomotion is even harder still, as locomotion is driven not by a single joint oscillating, but by multiple joints oscillating in synchronisation. This makes the required parameter space even smaller with respect to the total parameter space, and hence the genetic algorithm is less likely to discover good solutions.

12.2 Future work

This work raises many interesting questions:

- For spiking neurons, this works effectively bypasses the problem of spike coding by accessing the internals of a neuron to effectively communicate a continuous value. Would it be possible to use the genetic algorithm to co-evolve spike encoding and decoding functions for motor and sensory signals? Would it be possible to co-evolve the motor model instead of using a proportional derivative controller? Would it be possible to evolve the location of sensors within the geometry, and could sensors be parameterised so that the sensor functionality itself could be evolved?
- Wave generating functions are obviously very successful in driving low level motor behaviour for the locomotion task. What is the best scheme to integrate sensory data with these functions? Hornby presented one scheme which he termed “oscillator neurons”, where the model ensures that by default it will produce an oscillating output, but also enables input signals to somehow affect this output [206, p.63]. Lassabe used a stimulus-response action selection mechanism to classify inputs and select an output signal [255]. The table of mappings from stimulus to response was encoded in the genome and evolved, and successfully used to generate locomotion behaviour. Ngo used controllers that maintain mechanical equilibrium to move between different stances of a creature, but evolved both the parameters of the controllers and the stances together with a stimulus-response selection mechanism [319]. Hybrid schemes like this should be investigated, with performance being evaluated on some standardised benchmark, in order to determine which factors of the various hybridisation schemes are actually important.
- In this work the neuron functions were fixed and predetermined according to specified models (sigmoid, Taga, etc.). This presents the question as to which

neuron model to use for a particular evolutionary problem. Another option would be to encode the neuron function, or a set of functions, within the genome and make it subject to optimisation by the genetic algorithm. If successful, this would result in a description of a neuron model optimised along with specific neural networks and morphologies.

- The two levels of synchronisation between neurons in this work were globally synchronised or asynchronous. These represent the two extremes of timing — either every operation is synchronised, or none are. Possible hybrid schemes should be explored, and experiments carried out to find whether hybrid synchronisation affects the performance of evolved creatures on control tasks. One obvious avenue of research is to investigate whether the timing scheme can be encoded in the genome and co-evolved along with the rest of the creature.
- The two benchmark problems used in this work — pole balancing and creature locomotion — are well known and studied within the field of evolutionary robotics. This work has shown that, as benchmarks, these two problems are quite limited. Reasonably good solutions can be found for both through dumb oscillation. More standardised, open and rigorous benchmarks are required within the field to enable independent researchers to compare their controllers on the same tasks. It is not possible at the moment to compare different control systems from different published works as source code is not available, and researchers typically implement their own evolutionary robotics systems from scratch. This work has been guilty of the same, although it did at least use an open physics library rather than reimplementing it. It should not be necessary for researchers in the field to have to reimplement Sims's work again and again. The arrival of a standardised benchmark suite for evolutionary robotics would help to greatly advance the field.
- How can neuron output patterns be generated and stored within the genome? Parameterised sine waves and Bezier curves have been used [457], and so have sawtooth and square waves [397, 398]. Lassabe proposed a scheme whereby a large set of global waveforms were randomly generated initially, and then composed within the genome into unique sequences [255]. This approach is obviously not as adaptable as evolving the actual raw waveforms, but it was successfully used to generate locomotion behaviour.

- One avenue for possible future research in the co-evolution of morphologies and neural control would be to evolve the population size and number of generations over many more levels, and to establish how they affect performance of the evolved controllers, and whether there is a “sweet spot” in terms of particular fitness tasks and the amount of computational effort required, beyond which additional computational effort produces smaller gains on a level not proportionate to the computational resources invested. Another interesting possibility would be to examine the effect of using termination functions which end the genetic algorithm based upon criteria related to evolutionary progress rather than using a fixed number of generations. Dynamic population sizes have been studied by researchers in other fields, but the author is not aware of any research on dynamic populations across the space of evolved neural controllers (and morphologies) for simulated three-dimensional robots.
- It would be interesting to compare the performance of genetic algorithms and particle swarm optimisation on the combined evolution of morphology and control. Particle swarm optimisation extends the traditional genetic algorithm into a physics based space, where individual genes occupy one dimension, making a whole genotype multi-dimensional, and giving each genotype a mass. The algorithm accelerates individual genotypes in dimensions in which improvements are seen. This gives genotypes a velocity, with which they travel towards optimal solutions, overshoot, and then converge upon in a multi-dimensional spiral. The “swarm” comes from visualising many particles moving simultaneously — the group appears to rapidly converge and swarm about areas of high fitness in a manner similar to that of insects. It has previously been reported that on a neural network based robot control task particle swarm optimisation significantly outperformed genetic algorithms [42].
- There are a few ideas for further development of the software suite. The initial development phase is over, and the software is stable, but in order to roll it out to a wider audience some amount of further work may be desirable, such as writing a user’s guide. Some aspects of the software could be done better if it were being developed into a professional tool suite. Neural network activity is, at the moment, visualised by converting traces into PDF files. This is a very good solution for the purpose of writing a thesis, but sub-optimal for general research, as the resolution and user interface of PDF rendering software is limited. A

better solution would be to record the log in a standard format (e.g. LXT), where it can be visualised and analysed using standard software tools such as GtkWave.

- It would be useful to integrate the developed software with generic libraries for genetic algorithms, neural networks, and dynamical systems simulation. At the moment these kinds of functions are hard-coded into the source code. Whilst writing a simple genetic algorithm, or Euler integrator with neural differential equations, is not hard, any kind of software development is time consuming and prone to errors, and it would reduce the amount of duplication currently carried out by authors of similar systems. It would allow additional research to be carried out on search dynamics, as generic genetic algorithm libraries often implement a much more diverse collection of evolution strategies. A standard file representation, possibly based on XML, would allow creature models to be serialised and transferred between databases. Separating the evolutionary functions for neural network and morphology would allow plugging in new and promising algorithms such as the “neuro-evolution of augmenting topologies” [409].

Bibliography

- [1] Panagiotis Adamidis. Review of parallel genetic algorithms bibliography. Technical report, Aristotle University of Thessaloniki, Department of Electrical and Computer Engineering, 1994.
- [2] Anne M.R. Agur and Arthur F. Dalley. *Grant's Atlas of Anatomy*. Lippincott Williams & Wilkins, 2008.
- [3] R. Aharonov, L. Segev, I. Meilijson, and E. Ruppin. Localization of function via lesion analysis. *Neural Computation*, 15(4), 2003.
- [4] Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, and Peter Walter. *Molecular biology of the cell*. Garland, fourth edition, 2002.
- [5] S. Ando and H. Iba. Quantitative modeling of gene regulatory network-identifying the network by means of genetic algorithms. *Poster Session of Genome Informatics Workshop*, 2000.
- [6] Michael A. Arbib. *The Handbook of Brain Theory and Neural Networks*. The MIT Press, July 2003.
- [7] T. Arslan, E. Ozdemir, M. S. Bright, and D. H. Horrocks. Genetic synthesis techniques for low-power Digital Signal Processing circuits. In *Proceedings Of The IEE Colloquium On Digital Synthesis*, pages 7/1–7/5, London, UK, 1996. IEE.
- [8] STatistical Awareness and Teaching Support group. Testing the normality assumption of ANOVA models. 2009.
- [9] Yariv Bachar. Developing controllers for biped humanoid locomotion. Master's thesis, University of Edinburgh, 2004.
- [10] T. Bäck, F. Hoffmeister, and H. Schwefel. A survey of evolution strategies. In *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kauffman, July 1991.
- [11] Thomas Bäck. Optimal mutation rates in genetic search. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 2–8, San Mateo, CA, USA, 1993. Morgan Kaufmann.

- [12] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996.
- [13] Arunava Banerjee. On the phase-space dynamics of systems of spiking neurons. I: Model and experiments. *Neural Computation*, 13(1):161–193, 2001.
- [14] Nils Aall Barricelli. Symbiogenetic evolution processes realized by artificial methods. *Methodos*, 9:35–36, 1957.
- [15] David Basanta, Peter J. Bentley, Mark A. Miodownik, and Elizabeth A. Holm. Evolving cellular automata to grow microstructures. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Proceedings of the Sixth European Conference on Genetic Programming (EuroGP-2003)*, volume 2610 of *LNCS*, pages 1–10, Essex, UK, April 2003. Springer Verlag.
- [16] Roberto Battiti and Giampietro Tecchiolli. Training neural nets with the reactive tabu search. *IEEE Transactions on Neural Networks*, 6(5):1185–1200, September 1995.
- [17] Randall D. Beer. *Intelligence as adaptive behavior: an experiment in computational neuroethology*. Academic Press Professional, Inc., San Diego, CA, USA, 1990.
- [18] Randall D. Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3(4):469–509, 1995.
- [19] Randall D. Beer and John C. Gallagher. Evolving dynamical neural networks for adaptive behaviour. *Adaptive Behavior*, 1(1):91–122, 1992.
- [20] P. J. Bentley. *Generic evolutionary design of solid objects using a genetic algorithm*. PhD thesis, Division of Computing and Control Systems, School of Engineering, The University of Huddersfield, 1996.
- [21] P. J. Bentley. Exploring component-based representations- the secret of creativity by evolution? In I. C. Parmee, editor, *Proceedings of the Fourth International Conference on Adaptive Computing in Design and Manufacture (ACDM 2000)*, pages 161–172, April 2000.
- [22] P. J. Bentley and J. P. Wakefield. The evolution of solid object designs using genetic algorithms. In *Proceedings of the Conference on Applied Decision Technologies (ADT '95). Volume 2: Modern Heuristic Search Methods*, pages 391–402, Uxbridge, UK, April 1995. Unicom Seminars.
- [23] Peter Bentley and Sanjeev Kumar. Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakielka, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, pages 35–43, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

- [24] Dmitry Berenson, Nicolas Estevez, and Hod Lipson. Hardware evolution of analog circuits for in-situ robotic fault-recovery. In *2005 NASA / DoD Conference on Evolvable Hardware (EH 2005), 29 June - 1 July 2005, Washington, DC, USA*, pages 12–19. IEEE Computer Society, 2005.
- [25] Theodore W Berger, Ashish Ahuja, Spiros H Courellis, Samuel A Deadwyler, Gopal Erinjippurath, Gregory A Gerhardt, Ghassan Gholmeh, John J Granacki, Robert Hampson, Min Chi Hsiao, Jeffrey LaCoss, Vasilis Z Marmarelis, Patrick Nasiatka, Vijay Srinivasan, Dong Song, Armand R Tanguay, and Jack Wills. Restoring lost cognitive function. *Engineering in Medicine and Biology Magazine, IEEE*, 24(5):30–44, September 2005.
- [26] Karel P. Bergmann, Renate Scheidler, and Christian Jacob. Cryptanalysis using genetic algorithms. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1099–1100, New York, NY, USA, 2008. ACM.
- [27] Fabrice Bernhard and Renaud Keriven. Spiking neurons on GPUs. Technical Report 05-15, Centre d’Enseignement et de Recherche en Technologies de l’information et Systèmes, Ecole Nationale des Ponts et Chaussées, 77455 Marne, Paris, France, November 2005.
- [28] N. Bertschinger and T. Natschläger. Real-time computation at the edge of chaos in recurrent neural networks. *Neural Comput*, 16(7):1413–1436, July 2004.
- [29] A. Bhattacharjya and S. Liang. Power-law distributions in some random boolean networks. *Physical Review Letters*, 77(8), August 1996.
- [30] Sven Bilke and Fredrik Sjunnesson. Stability of the Kauffman model. *Physical Review E*, 65, December 2001.
- [31] Tobias Bickle and Lothar Thiele. A comparison of selection schemes used in genetic algorithms. Technical Report 11, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, 8092 Zurich, Switzerland, 1995.
- [32] E.J.W. Boers, H. Kuiper, B.L.M. Happel, and I.G. Sprinkhuizen-Kuyper. Designing modular artificial neural networks. In H.A. Wijshoff, editor, *Proceedings of Computing Science in The Netherlands*, pages 87–96, SION, Stichting Mathematisch Centrum, 1993.
- [33] J. Bongard and H. Lipson. Integrated design, deployment and inference for robot ecologies. In *Proceedings of Robosphere*, NASA Ames Research Center, CA USA, November 2004.
- [34] J. Bongard and C. Paul. Investigating morphological symmetry and locomotive efficiency using virtual embodied evolution. In J.-A. Meyer et al., editor, *From Animals to Animats: Proceedings of the Sixth International Conference on the Simulation of Adaptive Behaviour*, 2000.

- [35] J. C. Bongard and H. Lipson. Once more unto the breach: Automated tuning of robot simulation using an inverse evolutionary algorithm. In J. Pollack, M. Bedau, P. Husbands, T. Ikegami, and R.A Watson, editors, *Artificial Life IX: Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*. International Society for Artificial Life, The MIT Press, 2004.
- [36] Josh Bongard and Hod Lipson. Automated reverse engineering of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences (PNAS)*, 104(24):9943–9948, June 2007.
- [37] Josh C. Bongard. Evolving modular genetic regulatory networks. *Proceedings of the IEEE 2002 Congress on Evolutionary Computation (CEC2002)*, 2:1872–1877, 2002.
- [38] Josh C. Bongard. *Incremental approaches to the combined evolution of a robot's body and brain*. PhD thesis, Universität Zürich, 2003.
- [39] Josh C. Bongard and Chandana Paul. Making evolution an offer it can't refuse: Morphology and the extradimensional bypass. In *ECAL '01: Proceedings of the 6th European Conference on Advances in Artificial Life*, pages 401–412, London, UK, 2001. Springer-Verlag.
- [40] Josh C. Bongard and Rolf Pfeifer. Evolving complete agents using artificial ontogeny. *Morpho-functional Machines: The New Species (Designing Embodied Intelligence)*, pages 237–258, 2003.
- [41] Laurent Bonnasse-Gahot. Using genetic algorithms to evolve locomotion in artificial creatures. Technical Report D005, Département Informatique et Réseaux, Ecole Nationale Supérieure des Télécommunications, Paris, 2005.
- [42] Yvan Bourquin. Self organization of locomotion in modular robots. Master's thesis, University of Sussex, 2004.
- [43] George E. P. Box, William G. Hunter, and Stuart J. Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. Wiley-Interscience, June 1978.
- [44] Jérôme Braure. Participation to the construction of a salamander robot: exploration of the morphological configuration and the locomotion controller. Master's thesis, Biologically inspired robotics group, Swiss Federal Institute of Technology (EPFL), Lausanne, 2004.
- [45] D. Brogan and J. Hodgins. Group behaviors for systems with significant dynamics. *Autonomous Robots*, 4:137–153, 1997.
- [46] Jennifer C. Brookes, Filio Hartoutsiou, A. P. Horsfield, and A. M. Stoneham. Could humans recognize odor by phonon assisted tunneling? *Physical Review Letters*, 98(3):038101, 2007.

- [47] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1-3):139–159, January 1991.
- [48] T. G. Brown. On the nature of the fundamental activity of the nervous centres; together with an analysis of the conditioning of rhythmic activity in progression, and a theory of the evolution of function in the nervous system. *Journal of Physiology (London)*, 48:18–46, March 1914.
- [49] Jason Brownlee. The pole balancing problem. Technical Report 7-01, Centre for Intelligent Systems and Complex Processes, Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Victoria, Australia, 2005.
- [50] Gunnar Buason, Nicklas Bergfeldt, and Tom Ziemke. Brains, bodies, and beyond: Competitive co-evolution of robot controllers, morphologies and environments. *Genetic Programming and Evolvable Machines*, 6(1):25–51, 2005.
- [51] Jonathan B. Buckheit, Jonathan B. Buckheit, David L. Donoho, and David L. Donoho. Wavelab and reproducible research. pages 55–81. Springer-Verlag, 1995.
- [52] Anthony N. Burkitt. A review of the integrate-and-fire neuron model: I. Homogeneous synaptic input. *Biological Cybernetics*, 95(1):1–19, 2006.
- [53] G. Buttazzo. Artificial consciousness: Utopia or real possibility? *Computer*, 34(7):24–30, July 2001.
- [54] Santiago Ramón Y Cajal. *Textura del sistema nervioso del hombre y de los vertebrados. Translation: texture of the nervous system of man and the vertebrates*. Springer, 1999, 1899.
- [55] M. Capcarrere. Evolution of asynchronous cellular automata. In J.J.Merelo et al, editor, *The seventh Conference on Parallel Problem Solving From Nature, PPSN 2002*, pages 903–912. Springer-Verlag, September 2002.
- [56] Benedict Carey. H. m., an unforgettable amnesiac, dies at 82. *The New York Times*, January 2008.
- [57] Julio César Hernández Castro and Pedro Isasi Vi nuela. New results on the genetic cryptanalysis of TEA and reduced-round versions of XTEA. *New Generation Computing*, 23(3):233–243, 2005.
- [58] S. Chattopadhyay and N. Choudhary. Genetic algorithm based approach for low power combinational circuit testing. In *Proceedings of the 16th International Conference on VLSI Design*, pages 552–557, January 2003.
- [59] Saurabh Chaudhury, Krishna Teja Sistla, and Santanu Chattopadhyay. Genetic algorithm-based FSM synthesis with area-power trade-offs. *Integration, the VLSI Journal*, 42(3):376–384, 2009.

- [60] Nicolas Chaumont, Richard Egli, and Christoph Adami. Evolution of virtual catapults. In Luis Mateus Rocha, Larry S. Yaeger, Mark A. Bedau, Dario Floreano, Robert L. Goldstone, and Alessandro Vespignani, editors, *Artificial Life X: Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems*, pages 262–268. International Society for Artificial Life, The MIT Press, 2006.
- [61] Nicolas Chaumont, Richard Egli, and Christoph Adami. Evolving virtual creatures and catapults. *Artificial Life*, 13(2):139–57, 2007.
- [62] W. Chen, A.V. Rylyakov, V. Patel, J.E. Lukens, and K.K. Likharev. Rapid single flux quantum t-flip flop operating up to 770 GHz. *Applied Superconductivity, IEEE Transactions on*, 9(2):3212–3214, 1999.
- [63] C. Cherniak. Innateness and brain-wiring optimization: Non-genomic nativism. In A. Zilhao, editor, *Evolution, Rationality and Cognition*, volume 1. Routledge, November 2005.
- [64] C. Cherniak, M. Changizi, and D. Kang. Large-scale optimization of neuron arbors. *Physical Review E*, 59:6001–6009, May 1999.
- [65] C. Cherniak, Z. Mokhtarzada, and R. Rodriguez-Esteban. Neural wiring optimization. In Kaas et al., editor, *Evolution of Nervous Systems: A Comprehensive Reference*, volume 1. Academic Press Inc., November 2006.
- [66] C. Cherniak, Z. Mokhtarzada, R. Rodriguez-Esteban, and K. Changizi. Global optimization of cerebral cortex layout. *Proceedings of the National Academy Science U.S.A.*, 101:1081–1086, January 2004.
- [67] H. J. Chiel and R. D. Beer. The brain has a body: adaptive behavior emerges from interactions of nervous system, body and environment. *Trends Neuroscience*, 20(12):553–557, December 1997.
- [68] T. Chiueh and R.M. Goodman. Learning algorithms for neural networks with ternary weights. In *Proceedings of the First Annual Meeting of the International Neural Networks Society*, page 166, Boston, Massachusetts, USA, 1988.
- [69] Marcus Chown. Our world may be a giant hologram. *New Scientist*, 2691:24–27, January 2009.
- [70] Eric H. Chudler. *Neuroscience For Kids*. National Center for Research Resources and University of Washington, 2008.
- [71] Federica Ciocchetta and Jane Hillston. Process algebras in systems biology. In *Formal Methods for Computational Systems Biology*, volume 5016 of *Lecture Notes in Computer Science*, pages 265–312. Springer-Verlag, 2008.
- [72] Dave Cliff and Geoffrey F. Miller. Co-evolution of pursuit and evasion II: Simulation methods and results. In Pattie Maes, Maja J. Mataric, Jean-Arcady Meyer, Jordan B. Pollack, and Stewart W. Wilson, editors, *From animals to animats 4*, pages 506–515, Cambridge, MA, 1996. MIT Press.

- [73] Robert J. Collins and David R. Jefferson. An artificial neural network representation for artificial organisms. In H.-P. Schwefel and R. Männer, editors, *Parallel problem solving from nature: 1st Workshop, PPSN I*, pages 259–263, Berlin, 1991. Springer.
- [74] A. Colomni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In F. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life (ECAL)*, pages 134–142. MIT Press, Cambridge, Massachusetts, 1991.
- [75] Suzanne Corkin. What’s new with the amnesic patient H.M.? *Nature Reviews Neuroscience*, 3(2):153–160, February 2002.
- [76] F. Corno, P. Prinetto, M. Rebaudengo, and M. Reorda. A test pattern generation methodology for low power consumption. *16th IEEE VLSI Test Symposium*, pages 453–457, 1998.
- [77] Michael J. Crawley. *Statistics: an introduction using R*. Wiley, 2005.
- [78] J. P. Crutchfield and M. Mitchell. The evolution of emergent computation. In *Proceedings of the National Academy of Sciences*, volume 92, pages 10742–10746, November 1995.
- [79] James P. Crutchfield, Melanie Mitchell, and et al. The evolutionary design of collective computation in cellular automata. *Evolutionary Dynamics*, 2003.
- [80] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.
- [81] Ewa Dabrowska. *Language, mind and brain*. Edinburgh University Press, 2004.
- [82] Charles Darwin. *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life*. John Murray, 1859.
- [83] Rajarshi Das, James P. Crutchfield, Melanie Mitchell, and James E. Hanson. Evolving globally synchronized cellular automata. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 336–343, San Francisco, CA, 1995. Morgan Kaufmann.
- [84] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. In A. Kent and J. G. Williams, editors, *The Encyclopedia of Computer Science and Technology*, volume 38. Marcel Dekker, New York, February 1998.
- [85] Richard Dawkins. *The blind watchmaker*. Norton, 1985.
- [86] Richard Dawkins. *The ancestor’s tale : a pilgrimage to the dawn of evolution*. Houghton Mifflin, New York, 2004.
- [87] H. de Garis, F. A. Gers, M. Korkin, N. E. Nawa, and M. Hough. Evolving an optimal de/convolution function for the neural net modules of ATR’s CAM-Brain Machine (CBM). In *Proceedings of the International Joint Conference of Neural Networks*, 1999.

- [88] Hugo de Garis. The genetic programming of an artificial brain which grows/evolves at electronic speeds in a cellular automata machine. In *International Conference on Evolutionary Computation*, pages 337–339, 1994.
- [89] Hugo de Garis. CAM-Brain: The evolutionary engineering of a billion neuron artificial brain by 2001 which grows/evolves at electronic speeds inside a Cellular Automata Machine (CAM). In Eduardo Sanchez and Marco Tomassini, editors, *Towards Evolvable Hardware; The Evolutionary Engineering Approach*, pages 76–98, Berlin, 1996. Springer.
- [90] Hugo de Garis. What happened to the "CAM-Brain Machines" (CBMs)? May 2006.
- [91] Hugo de Garis, Andrzej Buller, Michael Korkin, Felix Gers, Norberto Eiji Nawa, and Michael Hough. ATR's artificial brain ("CAM-Brain") project: A sample of what individual "CoDi-1Bit" model evolved neural net modules can do with digital and analog I/O. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakielka, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, pages 13–17, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
- [92] Hugo de Garis, Michael Korkin, and Gary Fehr. The CAM-Brain Machine (CBM): An FPGA based tool for evolving a 75 million neuron artificial brain to control a lifesized kitten robot. *Autonomous Robots*, 10(3):235–249, 2001.
- [93] Hugo de Garis, Michael Korkin, Felix Gers, Norberto Eiji Nawa, and Michael Hough. "CAM-Brain" ATR's artificial brain project — an overview. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakielka, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
- [94] Edwin D. de Jong and Jordan B. Pollack. Learning the ideal evaluation function. In Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasa Jonoska, and Julian F. Miller, editors, *Genetic and Evolutionary Computation - GECCO 2003, Genetic and Evolutionary Computation Conference, Chicago, IL, USA, July 12-16, 2003. Proceedings, Part II*, volume 2724 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 2003.
- [95] H. de Jong. Modeling and simulation of genetic regulatory systems: a literature review. *Journal of Computational Biology*, 9(1):67–103, 2002.
- [96] Gonzalo G. de Polavieja. Errors drive the evolution of biological signalling to costly codes. *Journal of Theoretical Biology*, 214:657–664, February 2002.

- [97] F. Delcomyn. Neural basis of rhythmic behavior in animals. *Science*, 210:492–498, 1980.
- [98] F. Dellaert and R.D. Beer. Co-evolving body and brain in autonomous agents using a developmental model. Technical Report Technical Report CES-94-16, Department of computer engineering and science, Case Western Reserve University, Cleveland, OH 44106, 1994.
- [99] Frank Dellaert and R.D. Beer. A developmental model for the evolution of complete autonomous agents. In Pattie Maes et al., editor, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, 1996.
- [100] T. B. DeMarse and K. P. Dockendorf. Adaptive flight control with living neuronal networks on microelectrode arrays. In *Proceedings of the International Journal of Computation and Neural Networks 3*, pages 1548–1551, 2005.
- [101] Thomas B. Demarse, Daniel A. Wagenaar, Axel W. Blau, and Steve M. Potter. The neurally controlled animat: Biological brains acting with simulated bodies. *Auton. Robots*, 11(3):305–310, November 2001.
- [102] J.P. Demuth, T.D. Bie, J.E. Stajich, N. Cristianini, and M.W. Hahn. The evolution of mammalian gene families. *Public Library of Science (PLoS) Biology*, 1(e85), December 2006.
- [103] Z. J. Deng, N. Yoshikawa, J. A. Tiemo, S. R. Whiteley, and T. Van Duzer. Asynchronous circuits and systems in superconducting RSFQ technology. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 274–285, 1998.
- [104] R. Der, G. Mar, and F. Hesse. Let it roll - emerging sensorimotor coordination in a spherical robot. In Luis Mateus Rocha, Larry S. Yaeger, Mark A. Bedau, Dario Floreano, Robert L. Goldstone, and Alessandro Vespiagnani, editors, *Artificial Life X: Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems*, pages 192–198. International Society for Artificial Life, The MIT Press, 2006.
- [105] Advanced Micro Devices. AMD server/workstation solution provider roadmap Q3. 2006.
- [106] P. D'haeseleer, S. Liang, and R. Somogyi. Genetic network inference: from co-expression clustering to reverse engineering. *Bioinformatics*, 16(8):707–726, August 2000.
- [107] E. Di Paolo. Rhythmic and non-rhythmic attractors in asynchronous random boolean networks. *BioSystems*, 59(3):185–195, 2001.
- [108] Ezequiel A. Di Paolo. Searching for rhythms in asynchronous random boolean networks. In C. C. Maley and E. Boudreau, editors, *Proceedings of the Seventh International Conference on the Simulation and Synthesis of Living Systems (ALIFE7)*. International Society for Artificial Life, The MIT Press, 2000.

- [109] Edsger W. Dijkstra. On the cruelty of really teaching computing science. circulated privately, December 1988.
- [110] Jonathan Dinerstein, Nelson Dinerstein, and Hugo de Garis. Automatic multi-module neural network evolution in an artificial brain. In *Evolvable Hardware*, pages 283–286, 2003.
- [111] Norman Doidge. *The Brain That Changes Itself: Stories of Personal Triumph from the Frontiers of Brain Science*. Penguin, 2008.
- [112] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.
- [113] Sorin Draghici. On the capabilities of neural networks using limited precision weights. *Neural Networks*, 15(3):395–414, 2002.
- [114] Eric Drexler. *Engines of creation : the coming era of nanotechnology*. Anchor Books, 1986.
- [115] Daniel Drubach. *The Brain Explained*. Prentice Hall, 1999.
- [116] E Dubrova and M Teslenko. Compositional properties of random boolean networks. *Physical Review E*, 71, May 2005.
- [117] J. Duysens and H.W. Van de Crommert. Neural control of locomotion; the central pattern generator from cats to humans. *Gait & Posture*, 7(2):131–141, 1998.
- [118] Marc Ebner. Evolutionary design of objects using scene graphs. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward P. K. Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, 6th European Conference, EuroGP 2003, Essex, UK, April 14-16, 2003. Proceedings*, volume 2610 of *Lecture Notes in Computer Science*, pages 47–58. Springer, 2003.
- [119] Marc Ebner, Adrian Grigore, Alexander Heffner, and Jürgen Albert. Coevolution produces an arms race among virtual plants. In James A. Foster, Eve-lyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 316–325, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.
- [120] Peter Eggenberger. Evolving morphologies of simulated 3D organisms based on differential gene expression. In P. Husbands and I. Harvey, editors, *Proceedings of the 4th European Conference on Artificial Life (ECAL97)*. MIT Press, Cambridge, MA, 1997.
- [121] D. Ehninger and G. Kempermann. Neurogenesis in the adult hippocampus. *Cell and Tissue Research*, 331:243–250, January 2008.

- [122] Ö. Ekeberg. A combined neuronal and mechanical model of fish swimming. *Biological Cybernetics*, 69:363–374, 1993.
- [123] Ken Endo, Fuminori Yamasaki, Takashi Maeno, and Hiroaki Kitano. A method for co-evolving morphology and walking pattern of biped humanoid robot. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation, ICRA 2002, May 11-15, 2002, Washington, DC, USA*, pages 2775–2780. IEEE, 2002.
- [124] Ken Endo, Funinori Yamasaki, Takashi Maeno, and Hiroaki Kitano. Co-evolution of morphology and controller for biped humanoid robot. In Gal A. Kaminka, Pedro U. Lima, and Raúl Rojas, editors, *RoboCup 2002: Robot Soccer World Cup VI*, volume 2752 of *Lecture Notes in Computer Science*, pages 327–341. Springer, 2002.
- [125] P.S. Eriksson, E. Perfilieva, T. Björk-Eriksson, A.M. Alborn, C. Nordborg, D.A. Peterson, and F.H. Gage. Neurogenesis in the adult human hippocampus. *Nature Medicine*, 4:1313–1317, November 1998.
- [126] J. Fallon, S. Reid, R. Kinyamu, I. Opole, R. Opole, J. Baratta, M. Korc, T.L. Endo, A. Duong, G. Nguyen, M. Karkehabadhi, D. Twardzik, S. Patel, and S. Loughlin. In vivo induction of massive proliferation, directed migration, and differentiation of neural cells in the adult mammalian brain. *Proceedings of the National Academy of Science U.S.A.*, 97:14686–14691, December 2000.
- [127] Cynthia Ferrell. A comparison of three insect-inspired locomotion controllers. *Robotics and Autonomous Systems*, 16:135–159, 1995.
- [128] J. Fieres, A. Grübl, S. Philipp, K. Meier, J. Schemmel, and F. Schürmann. A platform for parallel operation of VLSI neural networks. In L.S. Smith, A. Hussain, and I. Aleksander, editors, *Proceedings of the Brain Inspired Cognitive Systems (BICS2004)*, 2004.
- [129] E. Fiesler, A. Choudry, and H. J. Caulfield. A weight discretization paradigm for optical neural networks. In *Proceedings of the International Congress on Optical Science and Engineering*, volume 1281, pages 164–173, Bellingham, Washington, 1990. The Society of Photo-Optical Instrumentation Engineers (SPIE).
- [130] Oxford Film and Television. Battle of the robots: The hunt for AI. October 2001.
- [131] Five.TV. The boy with a new head. January 2008.
- [132] Kurt W. Fleischer, David H. Laidlaw, Bena L. Currin, and Alan H. Barr. Cellular texture generation. *Computer Graphics*, 29(Annual Conference Series):239–248, 1995.
- [133] D. Floreano, Y. Epars, J.C. Zufferey, and C. Mattiussi. Evolution of Spiking Neural Circuits in Autonomous Mobile Robots. *International Journal of Intelligent Systems*, 21(9):1005–1024, 2006.

- [134] D. Floreano, N. Schoeni, G. Caprari, and J. Blynel. Evolutionary Bits'n'Spikes. In R. K. Standish, M. A. Beadau, and H. A. Abbass, editors, *Proceedings of the 8th International Conference on the Simulation and Synthesis of Living Systems (Alife 8)*. International Society for Artificial Life, The MIT Press, 2002.
- [135] Dario Floreano and Claudio Mattiussi. Evolution of spiking neural controllers for autonomous vision-based robots. In *ER '01: Proceedings of the International Symposium on Evolutionary Robotics From Intelligent Robotics to Artificial Life*, pages 38–61, London, UK, 2001. Springer-Verlag.
- [136] D. Fontaneto, E. A. Herniou, C. Boschetti, M. Caprioli, G. Melone, C. Ricci, and T. G. Barraclough. Independently evolving species in asexual bdelloid rotifers. *Public Library of Science (PLoS) Biology*, 5(4):e87, March 2007.
- [137] Stephanie Forrest and Melanie Mitchell. Relative building-block fitness and the building-block hypothesis. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 109–126. Morgan Kaufmann, San Mateo, CA, 1993.
- [138] N. Franceschini, J. M. Pichon, and C. Blanes. From insect vision to robot vision. *Philosophical Transactions of the Royal Society: Biological Sciences*, 337(1281):283–294, September 1992.
- [139] Peter L. Freddolino, Anton S. Arkhipov, Steven B. Larson, Alexander McPherson, and Klaus Schulten. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. *Structure*, 14(3):437–449, March 2006.
- [140] Peter Fromherz. Neuroelectronic interfacing: Semiconductor chips with ion channels, nerve cells, and brain. In Rainer Waser, editor, *Nanoelectronics and Information Technology*, pages 781–810. Wiley, 2003.
- [141] D. Frutiger, Josh C. Bongard, and Fumiya Iida. Iterative Product Engineering: Evolutionary Robot Design. *Proceedings of the Fifth International Conference on Climbing and Walking Robots*, pages 619–629, 2002.
- [142] A. E. Fry. Battle of the genomes: The struggle for survival in a microbial world. *International Journal Epidemiology*, April 2007.
- [143] Pablo Funes and Jordan B. Pollack. Computer evolution of buildable objects. In P. Husbands and I. Harvey, editors, *Fourth European Conference on Artificial Life*, pages 358–367, Cambridge, MA, 1997. MIT Press.
- [144] Pablo Funes and Jordan B. Pollack. Computer evolution of buildable objects for evolutionary design by computers. In Peter J. Bentley, editor, *Evolutionary Design by Computers*, pages 387–403. Morgan Kaufmann, San Francisco, CA, 1999.
- [145] Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gerhard Stegmann. An asynchronous low-power 80C51 microcontroller. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107, 1998.

- [146] John C. Gallagher. A neuromorphic paradigm for extrinsically evolved hybrid analog/digital device controllers: Initial explorations. In *Evolvable Hardware*, pages 48–58. IEEE Computer Society, 2001.
- [147] John C. Gallagher and Randall D. Beer. Evolution and analysis of dynamical neural networks for agents integrating vision, locomotion, and short-term memory. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakielka, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, pages 1273–1280, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
- [148] M. S. Gazzaniga. The split brain revisited. *Scientific American*, 279(1):50–55, July 1998.
- [149] N. Geard. Modelling gene regulatory networks: Systems biology to complex systems. ACCS technical report (draft), Australian Centre for Complex Systems, The University of Queensland., 2004.
- [150] N. Geard, K. Willadsen, and J. Wiles. Perturbation analysis: A complex systems pattern. In *The Second Australian Conference on Artificial Life (ACAL 2005)*. World Scientific in the Advances in Natural Computation series, 2005.
- [151] Felix A. Gers and Hugo de Garis. CAM-Brain: A new model for ATR’s cellular automata based artificial brain project. In Tetsuya Higuchi, Masaya Iwata, and Weixin Liu, editors, *Evolvable Systems: From Biology to Hardware, First International Conference, ICES 96, Tsukuba, Japan, October 7-8, 1996, Proceedings*, volume 1259 of *Lecture Notes in Computer Science*, pages 437–452. Springer, 1996.
- [152] Felix A. Gers, Hugo de Garis, and Michael Korkin. CoDi-1Bit: A simplified cellular automata based neuron model. In Jin-Kao Hao, Evelyne Lutton, Edmund M. A. Ronald, Marc Schoenauer, and Dominique Snyers, editors, *Artificial Evolution, Third European Conference, AE’97, Nîmes, France, 22-24 October 1997, Selected Papers*, volume 1363 of *Lecture Notes in Computer Science*, pages 315–334. Springer, 1998.
- [153] Carlos Gershenson. Introduction to random boolean networks. In M. Bedau, P. Husbands, T. Hutton, S. Kumar, and H. Suzuki, editors, *Workshop and Tutorial Proceedings, Ninth International Conference on the Simulation and Synthesis of Living Systems (ALife IX)*, September 2004.
- [154] Carlos Gershenson. Updating schemes in random boolean networks: Do they really matter? page 238, 2004.
- [155] W. Gerstner and W. K. Kistler. *Spiking neuron models*. Cambridge University Press, 2002.
- [156] A. Ghosh, S. Tstutsui, and H. Tanaka. Function optimization in nonstationary environment using steady state genetic algorithms with aging of individuals. In *IEEE World Congress on Computational Intelligence*, pages 666–671, 1998.

- [157] Yoav Gilad, Alicia Oshlack, Gordon K. Smyth, Terence P. Speed, and Kevin P. White. Expression profiling in primates reveals a rapid evolution of human transcription factors. *Nature*, 440(7081):242–245, March 2006.
- [158] R. Giuly. Jungleboogie: A system for studying brain-body evolution of virtual creatures. In R. Poli, S. Cagnoni, M. Keijzer, E. Costa, F. Pereira, G. Raidl, S. C. Upton, D. Goldberg, H. Lipson, E. de Jong, J. Koza, H. Suzuki, H. Sawai, I. Parmee, M. Pelikan, K. Sastry, D. Thierens, W. Stolzmann, P. L. Lanzi, S. W. Wilson, M. O'Neill, C. Ryan, T. Yu, J. F. Miller, I. Garibay, G. Holifield, A. S. Wu, T. Riopka, M. M. Meysenburg, A. W. Wright, N. Richter, J. H. Moore, M. D. Ritchie, L. Davis, R. Roy, and M. Jakielka, editors, *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, 26-30June 2004.
- [159] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.
- [160] David Goldberg. The race, the hurdle, and the sweet spot. In Peter J. Bentley, editor, *Evolutionary Design by Computers*, pages 105–118. Morgan Kaufmann, San Francisco, CA, 1999.
- [161] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Professional, January 1989.
- [162] Seth Copen Goldstein and Mihai Budiu. NanoFabrics: Spatial computing using molecular electronics. In *Proceedings of the 28th International Symposium on Computer Architecture 2001*, 2001.
- [163] F. Gomez and J. Schmidhuber. Evolving modular fast-weight networks for control. In W. Duch, J. Kacprzyk, E. Oja, and S. Zadrozy, editors, *15th International Conference on Artificial Neural Networks: Biological Inspirations - ICANN 2005, LNCS 3697*, volume 3697 of *Lecture Notes in Computer Science*, pages 383–389. Springer-Verlag Berlin Heidelberg, 2005.
- [164] T.G.W. Gordon and P.J. Bentley. On evolvable hardware. In S. Ovaska and L. Sytandera, editors, *Soft Computing in Industrial Electronics*, pages 279–323. Physica-Verlag Heidelberg, Germany, 2002.
- [165] S. Grillner. Neurobiological bases of rhythmic motor acts in vertebrates. *Science*, 228(4696):143–149, 1985.
- [166] S. Grillner, P. Wallen, and L. Brodin. Neuronal network generating locomotor behavior in lamprey: Circuitry, transmitters, membrane properties, and simulation. *Annual Review of Neuroscience*, 14:169–199, 1991.
- [167] Frederic Gruau. Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In J. D. Schaffer and D. Whitley, editors, *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN92)*, pages 55–74. The IEEE Computer Society Press, 1992.

- [168] Frederic Gruau. *Neural network synthesis using cellular encoding and the genetic algorithm*. PhD thesis, Ecole Normale Supérieure de Lyon, 1994.
- [169] P. Hagmann, M. Kurant, X. Gigandet, P. Thiran, V. J. Wedeen, R. Meuli, and J. P. Thiran. Mapping human whole-brain structural networks with Diffusion mri. *Public Library of Science (PLoS) ONE*, 2(7), 2007.
- [170] Patric Hagmann, Leila Cammoun, Xavier Gigandet, Reto Meuli, Christopher J. Honey, Van J. Wedeen, and Olaf Sporns. Mapping the structural core of human cerebral cortex. *Public Library of Science (PLoS) Biology*, 6(7):e159+, July 2008.
- [171] J. Hallam and A.J. Ijspeert. Using evolutionary methods to parameterize neural models: a study of the lamprey central pattern generator. In R.J. Duro, J. Santos, and M. Grana, editors, *Biologically inspired robot behavior engineering*, pages 119–142. Springer Verlag, Berlin, 2003.
- [172] Jennifer Hallinan and Janet Wiles. Evolving genetic regulatory networks using an artificial genome. In *APBC '04: Proceedings of the second conference on Asia-Pacific bioinformatics*, pages 291–296, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.
- [173] Verena Hamburger. Locomotion control of a quadruped robot based on engine primitive. Master's thesis, The Robotics Research Lab, Technische Universität Kaiserslautern, April 2005.
- [174] B. L. M. Happel and J. M. J. Murre. Design and evolution of modular neural network architectures. *Neural Networks*, pages 985–1004, 1994.
- [175] Z Haque, T.K. Lee, T Inoue, C Luk, S.U. Hasan, K Lukowiak, and N.I. Syed. An identified central pattern-generating neuron co-ordinates sensory-motor components of respiratory behavior in lymnaea. *European Journal Neuroscience*, pages 94–104, January 2006.
- [176] John M. Harlow. Passage of an iron rod through the head. *Neuropsychiatry and Clinincal Neuroscience*, 11:281–283, 1999.
- [177] Morten Hartmann and Pauline Catriona Haddow. Evolution of fault tolerant and noise robust designs. *IEE Proceedings-Computers and Digital Techniques*, 151(04):287–294, July 2004. Special issue on Evolvable Hardware.
- [178] L. H. Hartwell, J. J. Hopfield, S. Leibler, and A. W. Murray. From molecular to modular cell biology. *Nature*, 402(6761 Suppl), December 1999.
- [179] I. Harvey and T. Bossomaier. Time out of joint: Attractors in asynchronous random boolean networks. In *Proceedings of the Fourth European Conference on Articial Life*, pages 67–75. MIT Press., 1997.
- [180] I. Harvey, E. Vaughan, and E Di Paolo. Time and motion studies: The dynamics of cognition, computation and humanoid walking. In *Fourth Intl. Symp. on Human and Artificial Intelligence Systems: From Control to Autonomy.*, 2004.

- [181] B. Hasslacher and M. W. Tilden. Living machines. *Robotics and autonomous systems*, 15(1):143–169, July 1995.
- [182] Erin Hastings, Ratan Guha, , and Kenneth O. Stanley. NEAT particles: Design, representation, and animation of particle system effects. In *IEEE Symposium on Computational Intelligence and Games (CIG'07)*, 2007.
- [183] J. Hawks, E. T. Wang, G. M. Cochran, H. C. Harpending, and R. K. Moyzis. Recent acceleration of human adaptive evolution. *Proceedings of the National Academy of Science U.S.A.*, 104:20753–20758, December 2007.
- [184] Thomas Heimburg and Andrew D Jackson. On soliton propagation in biomembranes and nerves. *Proceedings of the National Academy of Sciences of the United States of America*, 102(28):9790–9795, July 2005.
- [185] H. Hellmich and H. Klar. An FPGA based simulation acceleration platform for spiking neural networks. In *Proceedings of the 2004 47th Midwest Symposium on Circuits and Systems*, volume 2, pages 389–392. IEEE Computer Society, July 2004.
- [186] Martin Hemberg. GENR8 - a design tool for surface generation. Master's thesis, Department of Physical Resource Theory, Chalmers University, Sweden, June 29 2001.
- [187] Martin Hemberg, U. M. O'Reilly, A. Menges, K. Jonas, M. Goncalves, and S. Fuchs. Exploring generative growth and evolutionary computation for architectural design. In P. Machado and J.J. Morelo, editors, *Art of Artificial Evolution*. Springer, 2006.
- [188] Martin Hemberg and Una-May O'Reilly. Extending grammatical evolution to evolve digital surfaces with genr8. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 299–308, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag.
- [189] Andreas V. M. Herz, Tim Gollisch, Christian K. Machens, and Dieter Jaeger. Modeling single-neuron dynamics and computations: A balance of detail and abstraction. *Science*, 314(5796):80–85, 2006.
- [190] Kazuo Hirai, Masato Hirose, Yuji Haikawa, and Toru Takenaka. The development of honda humanoid robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-98)*, pages 1321–1326, 1998.
- [191] Leigh R. Hochberg, Mijail D. Serruya, Gerhard M. Friehs, Jon A. Mukand, Maryam Saleh, Abraham H. Caplan, Almut Branner, David Chen, Richard D. Penn, and John P. Donoghue. Neuronal ensemble control of prosthetic devices by a human with tetraplegia. *Nature*, 442(7099):164–171, July 2006.
- [192] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117(4):500–544, August 1952.

- [193] Craig J. Hogan. Measurement of quantum fluctuations in geometry. *Physical Review D (Particles, Fields, Gravitation, and Cosmology)*, 77(10):104031, 2008.
- [194] J. H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.
- [195] SL Hooper. Central pattern generators. *Current Biology*, 10(5):176–192, March 2000.
- [196] W. Hordijk, J. Crutchfield, and M. Mitchell. Embedded-particle computation in evolved cellular automata. *Proceedings of Physics and Computation '96.*, 1996.
- [197] G. Hornby, S. Takamura, J. Yokono, O. Hanagata, T. Yamamoto, , and M. Fujita. Evolving robust gaits with aibo. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3040–3045, 2000.
- [198] G. S. Hornby. Functional scalability through generative representations: the evolution of table designs. *Environment and Planning B: Planning and Design*, 31(4):569–587, 2004.
- [199] G. S. Hornby, M. Fujita, S. Takamura, T. Yamamoto, and O. Hanagata. Autonomous evolution of gaits with the sony quadruped robot. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakielka, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, pages 1297–1304, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
- [200] Gregory Hornby. Generative representations for evolving families of designs. In Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasa Jonoska, and Julian F. Miller, editors, *Genetic and Evolutionary Computation - GECCO 2003, Genetic and Evolutionary Computation Conference, Chicago, IL, USA, July 12-16, 2003. Proceedings, Part II*, volume 2724 of *Lecture Notes in Computer Science*, pages 1678–1689. Springer, 2003.
- [201] Gregory Hornby, Hod Lipson, and Jordan B. Pollack. Evolution of generative design systems for modular physical robots. In *Proceedings of the 2001 IEEE International Conference on Robotics and Automation, ICRA 2001, May 21-26, 2001, Seoul, Korea*, pages 4146–4151. IEEE, 2001.
- [202] Gregory Hornby and Jordan B. Pollack. Creating high-level components with a generative representation for body-brain evolution. *Artificial Life*, 8(3):223–246, 2002.
- [203] Gregory S. Hornby. Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick

- Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1729–1736, Washington DC, USA, 25-29 June 2005. ACM Press.
- [204] Gregory S. Hornby and Jordan B. Pollack. Body-brain co-evolution using L-systems as a generative encoding. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 868–875, San Francisco, California, USA, 2001. Morgan Kaufmann.
 - [205] Gregory S. Hornby and Jordan B. Pollack. Evolving L-systems to generate virtual creatures. *Computers and Graphics*, 25(6):1041–1048, 2001.
 - [206] Gregory Scott Hornby. *Generative representations for evolutionary design automation*. PhD thesis, 2003. Advisor - Jordan B. Pollack.
 - [207] M. Hough, H. de Garis, M. Korkin, F. A. Gers, and N. E. Nawa. SPIKER : Analog waveform to digital spiketrain conversion in ATR's artificial brain "CAM-Brain" project. In *International Conference on Robotics and Artificial Life*, 1999.
 - [208] Feng-Hsiung Hsu. *Behind Deep Blue: building the computer that defeated the world chess champion*. Princeton University Press, Princeton, NJ, USA, 2004.
 - [209] Fatima T. Husain, Thomas P. Lozito, Antonio Ulloa, and Barry Horwitz. Investigating the neural basis of the auditory continuity illusion. *Journal of Cognitive Neuroscience*, 17:1275–1292, 2005.
 - [210] P. Husbands, T. Smith, N. Jakobi, and M. O’Shea. Better living through chemistry: Evolving gasnets for robot control. *Connection Science*, 10(3–4):185–210, 1998.
 - [211] M Hutzler, A Lambacher, B Eversmann, M Jenkner, R Thewes, and P Fromherz. High-resolution multi-transistor array recording of electrical field potentials in cultured brain slices. *Journal Neurophysiology*, May 2006.
 - [212] V. Ila, J. Batlle, X. Cufí, and R. Garcia. Recent trends in FPAA devices. In Panos Liatsis, editor, *Recent Trends in Multimedia Information Processing*, pages 180–190, Manchester, November 2002. International Workshop on Systems, Signals and Image Processing, World Scientific.
 - [213] T. E. Ingerson and R. L. Buvel. Structure in asynchronous cellular automata. *Physica D: Nonlinear Phenomena*, 10:59–68, January 1984.
 - [214] F. Jacob. Evolution and tinkering. *Science*, 196(4295):1161–1166, June 1977.

- [215] N. Jakobi. The minimal simulation approach to evolutionary robotics. In T. Gomi, editor, *Evolutionary Robotics - From Intelligent Robots to Artificial Life (ER'98)*. AAI Books, 1998.
- [216] N. Jakobi. *Minimal simulations for evolutionary robotics*. PhD thesis, Centre for Research in Cognitive Science, University of Sussex, 1998.
- [217] Mariusz H. Jakubowski, Ken Steiglitz, and Richard Squier. Computing with solitons: a review and prospectus. In *Collision-based computing*, pages 277–297. Springer-Verlag, London, UK, 2002.
- [218] A. Jarosch and J. F. Leber. Opensim: A flexible distributed neural network simulator with automatic interactive graphics. *Neural Networks*, 10(4):693–703, June 1997.
- [219] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang. Evolution as a theme in artificial life: The genesys/tracker system. In J. Farmer C. Langton, C. Taylor and S. Rasmussen, editors, *Artificial Life II*, volume 10, pages 549–578. Addison-Wesley, Reading, MA, 1991.
- [220] Colin G. Johnson and Juan Jesus Romero Cardalda. Evolutionary computing in visual art and music. *Leonardo*, 35(2):175–184, April 2002.
- [221] Jet Propulsion Laboratory JPL. *NASA facts: Mars exploration rover*. National Aeronautics and Space Administration, California Institute of Technology, Pasadena, CA 91109, 2004.
- [222] Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell. *Principles of neural science*. McGraw-Hill Medical, January 2000.
- [223] Stuart A. Kauffman. Metabolic stability and epigenesis in randomly connected genetic nets. *Journal of Theoretical Biology*, 22:437–467, 1968.
- [224] Stuart A. Kauffman. *The origins of order: self-organization and selection in evolution*. Oxford University Press, May 1993.
- [225] Stuart A. Kauffman. *At home in the universe*. Oxford University Press, New York; Oxford, 1995.
- [226] R.A. Kaul, I.S. Naweed, and P. Fromherz. Neuron-semiconductor chip with chemical synapse between identified neurons. *Physical Review Letters*, 92, 2004.
- [227] Alon Keinan, Ben Sandbank, Claus C. Hilgetag, Isaac Meilijson, and Eytan Ruppin. Fair attribution of functional contribution in artificial and biological networks. *Neural Computation*, 16(9):1887–1915, 2004.
- [228] S. Kelso. *Dynamic Patterns: the self-organization of brain and behavior*. The MIT Press, 1995.

- [229] G.D. Kendall and T.J. Hall. Performing fundamental image processing operations using quantised neural networks. In *International Conference on Image Processing and its Applications*, pages 226–229, April 1992.
- [230] Didier Keymeulen, Gerhard Klimeck, Ricardo Zebulum, Yili Jin, Adrian Stoica, and Carlos Salazar-Lazaro. Ehwpack: A parallel software/hardware environment for evolvable hardware. In Darrell Whitley, editor, *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 162–169, Las Vegas, Nevada, USA, July 2000.
- [231] A. H. Khan and E. L. Hines. Integer-weight neural nets. *Electronics Letters*, 30(15):1237–1238, 1994.
- [232] H. Kilic and M. Güler. On the design of minimal topology non-uniform cellular automata for the binary sequence generation problem. In P. Husbands and I. Harvey, editors, *Proceedings of the 4th European Conference on Artificial Life (ECAL97)*. MIT Press, Cambridge, MA, 1997.
- [233] Werner M. Kistler, Wulfram Gerstner, and J. Leo van Hemmen. Reduction of the Hodgkin-Huxley equations to a single-variable threshold model. *Neural Computation*, 9(5):1015–1045, 1997.
- [234] Jon Klein. Breve: a 3D environment for the simulation of decentralized systems and artificial life. In *ICAL 2003: Proceedings of the eighth international conference on Artificial life*, pages 329–334, Cambridge, MA, USA, 2003. MIT Press.
- [235] Konstantin Klemm and Stefan Bornholdt. Stable and unstable attractors in boolean networks. *Physical Review E*, 72, 2005.
- [236] Jérôme Kodjabachian and Jean-Arcady Meyer. Evolution and development of control architectures in animats. *Robotics and Autonomous Systems*, 16(2-4):161–182, 1995.
- [237] Jérôme Kodjabachian and Jean-Arcady Meyer. Evolution and development of modular control architectures for 1D locomotion in six-legged animats. *Connection Science*, 10(3-4):211–237, 1998.
- [238] Jerome Kodjabachian and Jean-Arcady Meyer. Evolution and development of neural controllers for locomotion, gradient-avoidance, and obstacle-avoidance in artificial insects. *IEEE Transactions on Neural Networks*, 9(5):796–812, 1998.
- [239] Maciej Komosinski. Framsticks manual. Technical Report RA-011/00, Poznan University of Technology, Institute of Computing Science, 2000.
- [240] Maciej Komosinski. The world of Framsticks: Simulation, evolution, interaction. In *VW '00: Proceedings of the Second International Conference on Virtual Worlds*, pages 214–224, London, UK, 2000. Springer-Verlag.

- [241] Maciej Komosinski. The Framsticks system: versatile simulator of 3D agents and their evolution. *Kybernetes: The International Journal of Systems & Cybernetics*, 32:156–173, 2003.
- [242] Maciej Komosinski and Adam Rotaru-Varga. Comparison of different genotype encodings for simulated 3D agents. *Artificial Life Journal*, 7(4):395–418, Fall 2001.
- [243] Maciej Komosinski and Szymon Ulatowski. Framsticks: towards a simulation of a nature-like world, creatures and evolution. In Dario Floreano, Jean-Daniel Nicoud, and Francesco Mondada, editors, *Advances in Artificial Life*, volume 1674 of *Lecture Notes in Artificial Intelligence*, pages 261–265. Springer-Verlag, 1999.
- [244] M. Korkin, N. Nawa, and H. de Garis. A 'spike interval information coding' representation for ATR's CAM-Brain Machine (CBM). In *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES'98)*. Springer-Verlag, September 1998.
- [245] Michael Kositsky, Amir Karniel, Simon Alford, Karen M. Fleming, and Fernando A. Mussa-Ivaldi. Dynamical dimension of a hybrid neurorobotic system. *IEEE Transactions on neural systems and rehabilitation engineering*, 11(2):155–159, June 2003.
- [246] J. Kovačević. How to encourage and publish reproducible research. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume IV, pages 1273–1276, April 2007.
- [247] John R. Koza. Genetically breeding populations of computer programs to solve problems in artificial intelligence. In *Proceedings of the Second International Conference on Tools for AI, Herndon, Virginia, USA*, pages 819–827. IEEE Computer Society Press, Los Alamitos, CA, USA, 6-9 1990.
- [248] John R. Koza, Forrest H Bennett III, David Andre, Martin A. Keane, and Frank Dunlap. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on Evolutionary Computation*, 1(2):109–128, 1997.
- [249] J Kruger and F Aiple. Multimicroelectrode investigation of monkey striate cortex: spike train correlations in the infragranular layers. *Journal of Neurophysiology*, 60(2):798–828, August 1988.
- [250] Hiroyuki Kurata, Takahiro Inoue, Yoshiyuki Sumida, Shin Tanaka, and Takashi Ohashi. Simulation and system analysis of gene regulatory networks using the two-phase partition method. *Genome Informatics*, 12:286–287, 2001.
- [251] H.K. Kwan and C.Z. Tang. Designing multilayer feedforward neural networks using simplified sigmoid activation functions and one-powers-of-two weights. *Electronics Letters*, 28(25):2343–2345, 1992.

- [252] H.K. Kwan and C.Z. Tang. Multiplierless multilayer feedforward neural network design using quantised neurons. *Electronics Letters*, 38(13):645–646, 2002.
- [253] Christine Laine, Steven N Goodman, Michael E Griswold, and Harold C Sox. Reproducible research: moving toward research the public can really trust. *Annals of Internal Medicine*, 146(6):450–3, 2007.
- [254] Chris G. Langton. Computation at the edge of chaos: phase transitions and emergent computation. In *CNLS '89: Proceedings of the ninth annual international conference of the Center for Nonlinear Studies on Self-organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks on Emergent computation*, pages 12–37, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co.
- [255] Nicolas Lassabe, Hervée Luga, and Yves Duthen. A New Step for Evolving Creatures. In *IEEE-ALife'07, Honolulu, Hawaii, 01/04/2007-05/04/2007*, pages 243–251, <http://www.ieee.org/>, April 2007. IEEE.
- [256] Steven Laureys, Melanie Boly, and Pierre Maquet. Tracking the recovery of consciousness from coma. *Clinical Investigation*, 116(7):1823–1825, July 2006. Comment.
- [257] Wei-Po Lee, John Hallam, and Henrik Hautop Lund. A hybrid GP/GA approach for co-evolving controllers and robot bodies to achieve fitness-specified tasks. In *International Conference on Evolutionary Computation*, pages 384–389, 1996.
- [258] Robert Legenstein and Wolfgang Maass. 2007 special issue: Edge of chaos and prediction of computational performance for neural circuit models. *Neural Networks*, 20(3):323–334, 2007.
- [259] Erez Lieberman-Aiden, Nynke L. van Berkum, Louise Williams, Maxim Imakaev, Tobias Ragoczy, Agnes Telling, Ido Amit, Bryan R. Lajoie, Peter J. Sabo, Michael O. Dorschner, Richard Sandstrom, Bradley Bernstein, M. A. Bender, Mark Groudine, Andreas Gnirke, John Stamatoyannopoulos, Leonid A. Mirny, Eric S. Lander, and Job Dekker. Comprehensive mapping of long-range interactions reveals folding principles of the human genome. *Science*, 326(5950):289–293, October 2009.
- [260] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, March 1968.
- [261] H. Lipson. Evolutionary robotics and open-ended design automation. In Yoseph Bar-Cohen, editor, *Biomimetics: Biologically Inspired Technologies*. CRC Press, 2006.
- [262] H Lipson and J B Pollack. Automatic design and manufacture of robotic life-forms. *Nature*, 406(6799):974–8, 2000.

- [263] Hod Lipson and Josh Bongard. An exploration-estimation algorithm for synthesis and analysis of engineering systems using minimal physical testing. In *Proceedings of the ASME Design Automation Conference (DAC04)*. ACM Press, 2004.
- [264] Jason D. Lohn, Gregory Hornby, and Derek S. Linden. Evolution, re-evolution, and prototype of an X-band antenna for NASA's space technology 5 mission. In Juan Manuel Moreno, Jordi Madrenas, and Jordi Cosp, editors, *ICES*, volume 3637 of *Lecture Notes in Computer Science*, pages 205–214. Springer, 2005.
- [265] Jason D. Lohn, Gregory S. Hornby, and Derek S. Linden. An evolved antenna for deployment on NASA's space technology 5 mission. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund K. Burke, Paul J. Darwen, Dipankar Dasgupta, Dario Floreano, James A. Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andrew M. Tyrrell, editors, *Genetic and Evolutionary Computation - GECCO 2004, Genetic and Evolutionary Computation Conference, Seattle, WA, USA, June 26-30, 2004, Proceedings, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 26–30. Springer, June 2004.
- [266] Heitor S. Lopes and Mauricio Perretto. Reconstruction of phylogenetic trees using the ant colony optimization paradigm. In Natália F. Martins, Maria Emilia Telles Walter, Guilherme P. Telles, and Marcelo M. Brigido, editors, *III Brazilian Workshop on Bioinformatics, October 20-22, 2004, Brasília, Distrito Federal, Brazil*, pages 49–56, 2004.
- [267] T. Lundin, E. Fiesler, and P. Moerland. Connectionist quantization functions. In *Proceedings of the '96 SIPAR-Workshop on Parallel and Distributed Computing*, pages 33–36. Scientific and Parallel Computing Group, University of Geneva, 1996.
- [268] S. Luo, M. Kezunovic, and D.R. Sevcik. Locating faults in the transmission network using sparse field measurements, simulation data and genetic algorithms. *Electric Power Systems Research*, 71(2), October 2004.
- [269] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Electronic Colloquium on Computational Complexity (ECCC)*, 3(31), 1996.
- [270] M MacKay-Lyons. Central pattern generation of locomotion: a review of the evidence. *Physical Therapy*, 82(1):69–83, 2002.
- [271] Siavash Haroun Mahdavi and Peter J. Bentley. An evolutionary approach to damage recovery of robot motion with muscles. In Wolfgang Banzhaf, Thomas Christaller, Peter Dittrich, Jan T. Kim, and Jens Ziegler, editors, *Advances in Artificial Life, 7th European Conference, ECAL 2003, Dortmund, Germany, September 14-17, 2003, Proceedings*, volume 2801 of *Lecture Notes in Computer Science*, pages 248–255. Springer, 2003.

- [272] James Mallet. *Trends in Ecology and Evolution*, 20(5):229–237, 2005.
- [273] James Mallet. Hybridization, ecological races, and the nature of species: empirical evidence for the ease of speciation. *Philosophical Transactions of the Royal Society B — Biological Sciences*, 363:2971–2986, 2008.
- [274] E. Malone and H. Lipson. Fab@home: The personal desktop fabricator kit. In *Proceedings of the 17th Solid Freeform Fabrication Symposium*, Austin TX, August 2006.
- [275] P. Mandik. Varieties of representation in evolved and embodied neural networks. *Biology and Philosophy*, 18:95–130(36), January 2003.
- [276] D. Marbach and A.J. Ijspeert. Co-evolution of configuration and control for homogenous modular robots. In F. Groen et al., editor, *Proceedings of the Eighth Conference on Intelligent Autonomous Systems (IAS8)*, pages 712–719. IOS Press, 2004.
- [277] Daniel Marbach, Claudio Mattiussi, and Dario Floreano. Bio-mimetic evolutionary reverse engineering of genetic regulatory networks. In *5th European Conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics (EvoBIO 2007)*, pages 155–165, 2007. Editors: E. Marchiori J. H. Moore J. C. Rajapakse (Eds.) Publisher: Springer-Verlag Berlin Heidelberg.
- [278] M. Marchesi, G. Orlandi, F. Piazza, L. Pollonara, and A. Uncini. Multilayer perceptrons with discrete weights. In *Proceedings of International Joint Conference on Neural Networks*, volume 2, pages 623–630, San Diego, 1990.
- [279] Eve Marder, Dirk Bucher, David J. Schulz, and Adam L. Taylor. Invertebrate central pattern generation moves along. *Current Biology*, 15:685–699, September 2005.
- [280] Davide Marocco, Angelo Cangelosi, and Stefano Nolfi. The emergence of communication in evolutionary robots. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 361(1811):2397–2421, 2003.
- [281] G. Massera, S. Nolfi, and A. Cangelosi. Evolving a simulated robotic arm able to grasp objects. In A. Cangelosi et al., editor, *Modeling Language, Cognition and Action: Proceeding of the Ninth Neural Computation and Psychology Workshop Progress in Neural Processing 16*, pages 203–207, 2005.
- [282] Gianluca Massera, Angelo Cangelosi, and Stefano Nolfi. Developing a reaching behaviour in an simulated anthropomorphic robotic arm through an evolutionary technique. In Luis Mateus Rocha, Larry S. Yaeger, Mark A. Bedau, Dario Floreano, Robert L. Goldstone, and Alessandro Vespignani, editors, *Artificial Life X: Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems*, pages 234–240. International Society for Artificial Life, The MIT Press, 2006.

- [283] J. Mazoyer. Computations on cellular automata. In Delorme M. and Mazoyer J., editors, *Cellular Automata: a parallel model*, pages 303–319. Kluwer Academic Publishers, 1998.
- [284] P. Mazumder and E.M. Rudniuck. *Genetic algorithms for VLSI design layout and test automation*. Prentice-Hall, 1999.
- [285] Jon McCormack. Aesthetic evolution of L-systems revisited. In Günther R. Raidl, Stefano Cagnoni, Jürgen Branke, David Corne, Rolf Drechsler, Yaochu Jin, Colin G. Johnson, Penousal Machado, Elena Marchiori, Franz Rothlauf, George D. Smith, and Giovanni Squillero, editors, *Applications of Evolutionary Computing, EvoWorkshops 2004: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoMUSART, and EvoSTOC, Coimbra, Portugal, April 5-7, 2004, Proceedings*, volume 3005 of *Lecture Notes in Computer Science*, pages 477–488. Springer, 2004.
- [286] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.
- [287] Carver Mead. *Analog VLSI and neural systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
- [288] Gregor Johann Mendel. Versuche über pflanzen-hybriden. Translation: experiments on plant hybridization. In *Proceedings of the Natural History Society of Brunn*, 1866.
- [289] L. Mendoza, D. Thieffry, and Er Alvarez-Buylla. Genetic control of flower morphogenesis in arabidopsis thaliana: a logical analysis. *Bioinformatics*, 15(7):593–606, July 1999.
- [290] Ralph C. Merkle. How many bytes in human memory? *Foresight Update*, 4, 1988.
- [291] Ralph C. Merkle. Energy limits to the computational power of the human brain. *Foresight Update*, 6, 1989.
- [292] William C. Messner and Dawn M. Tilbury. *Control tutorials for MATLAB and Simulink: a web-based approach*. Addison-Wesley, 1998.
- [293] J.-A. Meyer. Evolutionary approaches to walking and higher-level behaviors in 6-legged animats. In Gomi, editor, *Evolutionary Robotics II: From Intelligent Robots to Artificial Life (ER'98)*. AAAI Books, 1998.
- [294] Olivier Michel. Webots: Symbiosis between virtual and real mobile robots. In *VW '98: Proceedings of the First International Conference on Virtual Worlds*, pages 254–263, London, UK, 1998. Springer-Verlag.
- [295] T. Miconi and A. Channon. A virtual creatures model for studies in artificial evolution. 2005.

- [296] Thomas Miconi. *The road to everywhere: Evolution, complexity and progress in natural and artificial systems*. PhD thesis, University of Birmingham, 2008.
- [297] Thomas Miconi and Alastair Channon. Analysing co-evolution among artificial 3D creatures. In El-Ghazali Talbi, Pierre Liardet, Pierre Collet, Evelyne Lutton, and Marc Schoenauer, editors, *Artificial Evolution, 7th International Conference, Evolution Artificielle, EA 2005, Lille, France, October 26-28, 2005, Revised Selected Papers*, volume 3871 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 2006.
- [298] Thomas Miconi and Alastair Channon. An improved system for artificial creatures evolution. In Luis Mateus Rocha, Larry S. Yaeger, Mark A. Bedau, Dario Floreano, Robert L. Goldstone, and Alessandro Vespignani, editors, *Artificial Life X: Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems*, pages 255–261. International Society for Artificial Life, The MIT Press, 2006.
- [299] Julian Francis Miller. Evolving a self-repairing, self-regulating, French flag organism. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund K. Burke, Paul J. Darwen, Dipankar Dasgupta, Dario Floreano, James A. Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andrew M. Tyrrell, editors, *Genetic and Evolutionary Computation - GECCO 2004, Genetic and Evolutionary Computation Conference, Seattle, WA, USA, June 26-30, 2004, Proceedings, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 129–139. Springer, June 2004.
- [300] G.L. Ming and H. Song. Adult neurogenesis in the mammalian central nervous system. *Annual Review of Neuroscience*, 28:223–250, 2005.
- [301] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, Mass., 1969.
- [302] M. Mitchell, J. Crutchfield, and R. Das. Evolving cellular automata with genetic algorithms: A review of recent work. In *Proceedings of the First International Conference on Evolutionary Computation and its Applications (EvCA'96)*, 1996., 1996.
- [303] Melanie Mitchell. Computation in cellular automata. In Tino Gramss, Stefan Bornholdt, Michael Gross, Melanie Mitchell, and Thomas Pellizzari, editors, *Non-Standard Computation: molecular computation, cellular automata, evolutionary algorithms, quantum computers*. Wiley-VCH, 1998.
- [304] Melanie Mitchell, James P. Crutchfield, and Peter T. Hraber. Dynamics, computation, and the “edge of chaos”: a re-examination. In *Complexity: metaphors, models, and reality*, pages 497–513. Perseus Books, Cambridge, MA, USA, 1999.

- [305] Melanie Mitchell, Peter T. Hraber, and James P. Crutchfield. Revisiting the edge of chaos: Evolving cellular automata to perform computations. *Complex Systems*, 7:89–130, 1993.
- [306] Tom M. Mitchell. *Machine learning*. McGraw Hill, New York, 1997.
- [307] Sechi Miyakoshi, Gentaro Taga, Yasuo Kuniyoshi, and Akihiko Nagakubo. Three dimensional bipedal stepping motion using neural oscillators — towards humanoid motion in the real world. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 84–89, Piscataway, NJ, 1998. IEEE Computer Society.
- [308] P. Moerland and E. Fiesler. Hardware-friendly learning algorithms for neural networks: An overview. In *Proceedings of the Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems: MicroNeuro'96*, pages 117–124, 10622 Los Vaqueros Circle, Los Alamitos, CA 90720, 1996. IEEE Computer Society Press.
- [309] P. D. Moerland and E. Fiesler. Neural network adaptations to hardware implementations. In E. Fiesler and R. Beale, editors, *Handbook of Neural Computation*, pages E1.2:1–13. Institute of Physics Publishing and Oxford University Publishing, New York, 1997. IDIAP-RR 97-17.
- [310] Edward Moore. *Sequential machines*. Addison-Wesley, 1964.
- [311] Hans Moravec. When will computer hardware match the human brain? *Journal of Evolution and Technology*, 1, 1998.
- [312] Hans Moravec. Rise of the robots. *Scientific American*, 281(6):124–135, December 1999.
- [313] Thomas Hunt Morgan. *A critique of the theory of evolution*. Princeton University Press, 1916.
- [314] Anthony Mouraud, Didier Puzenat, and Helene Paugam-Moisy. Damned: A distributed and multithreaded neural event-driven simulation framework. In *Proceedings of the IASTED International Conference on parallel and distributed computing and networks*, 2006.
- [315] U. Muller. Fast neural net simulation with a DSP processor array. *IEEE Transactions on Neural Networks*, 6(1):203–213, 1995.
- [316] Norberto Eiji Nawa, Michael Korkin, and Hugo de Garis. ATR's CAM-Brain project: The evolution of large-scale recurrent neural network modules. In H.R. Arabnia, editor, *PDPTA'98 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, Las Vegas, July 1998. CSREA Press. Four volumes.
- [317] H. H. Newman. Twin and triplet chick embryos. *Journal of Heredity*, 31(9):370–378, 1940.

- [318] Thomas C. Neylan. Frontal lobe function: Mr. Phineas Gage's famous injury. *Neuropsychiatry and Clinical Neuroscience*, 11:280–281, 1999.
- [319] J. Thomas Ngo and Joe Marks. Physically realistic motion synthesis in animation. *Evolutionary Computation*, 1(3):235–268, 1993.
- [320] S. Nolfi and D. Floreano. Co-evolving predator and prey robots: Do 'arm races' arise in artificial evolution? *Artificial Life*, 4:311–335, 1998.
- [321] Stefano Nolfi and Dario Floreano. Synthesis of autonomous robots through evolution. *Trends in Cognitive Science*, 6(1):31–36, 2002.
- [322] Gabriela Ochoa. On genetic algorithms and Lindenmayer systems. In *PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, pages 335–344, London, UK, 1998. Springer-Verlag.
- [323] Randall C. O'Reilly and Yuko Munakata. *Computational explorations in cognitive neuroscience: understanding the mind by simulating the brain*. MIT Press, 2000.
- [324] S. Osawa, T. H. Jukes, K. Watanabe, A. Muto, and T. H. Jukes. Recent evidence for evolution of the genetic code. *Microbiology Review*, 56:229–264, March 1992.
- [325] F. Pasemann and U. Dieckmann. Evolved neurocontrollers for polebalancing. In *Proceedings IWANN'97*, Lanzarote, Spain, June 1997. Lecture Notes in Computer Science. Berlin: Springer-Verlag.
- [326] Priyadarsan Patra, Donald S. Fussell, and Stanislav Polonsky. Delay insensitive logic for RSFQ superconductor technology. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 42–53. IEEE Computer Society Press, April 1997.
- [327] Chandana Paul. Sensorimotor control of biped locomotion. *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems*, 13(1):67–80, 2005.
- [328] Chandana Paul and Josh C. Bongard. The road less travelled: Morphology in the optimization of biped robot locomotion. In *Proceedings of the International Conference on Intelligent Robots and Systems*, Maui, Hawaii, October 2001. IEEE/RSJ.
- [329] B. Pearlmutter. Dynamic recurrent neural networks. Technical report, 1990.
- [330] Roger Penrose. *The Emperor's new mind: concerning computers, minds, and the laws of physics*. Oxford University Press, Inc., New York, NY, USA, 1989.
- [331] Ferdinand Peper, Jia Lee, Susumu Adachi, and Shinro Mashiko. Laying out circuits on asynchronous cellular arrays: a step towards feasible nanocomputers? *Nanotechnology*, 14(4):469–485, 2003.

- [332] Rolf Pfeifer. Morphological computation: Connecting brain, body, and environment. In *Proceedings of the Second International Workshop on Biologically Inspired Approaches to Advanced Information Technology (BioADIT06)*, pages 2–3, 2006.
- [333] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz A. Barroso. Failure trends in a large disk drive population. In *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, page 2, Berkeley, CA, USA, 2007. USENIX Association.
- [334] Vincenzo Piuri. The use of the electrical simulator SPICE for behavioral simulation of artificial neural networks. In *ANSS '91: Proceedings of the 24th annual symposium on Simulation*, pages 18–29, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.
- [335] Vassilis P. Plagianakos, George D. Magoulas, and Michael N. Vrahatis. Evolutionary training of hardware realizable multilayer perceptrons. *Neural Computing and Applications*, 15(1):33–40, 2006.
- [336] Vassilis P. Plagianakos and Michael N. Vrahatis. Training neural networks with threshold activation functions and constrained integer weights. In *Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN 2005)*, pages 161–166, 2000.
- [337] Vassilis P. Plagianakos and Michael N. Vrahatis. Parallel evolutionary training algorithms for “hardware-friendly” neural networks. *Natural Computing: an international journal*, 1(2-3):307–322, 2002.
- [338] Jordan Pollack, Hod Lipson, Pablo Funes, Sevan Ficici, and Greg Hornby. Co-evolutionary robotics. In *EH '99: Proceedings of the 1st NASA/DOD workshop on Evolvable Hardware*, page 208, Washington, DC, USA, 1999. IEEE Computer Society.
- [339] Jordan B. Pollack, Gregory S. Hornby, Hod Lipson, and Pablo Funes. Computer creativity in the automatic design of robots. *Leonardo, Journal for the International Society for Arts Sciences and Technology*, 36(2):115–121, 2003.
- [340] Jordan B. Pollack, Hod Lipson, Sevan Ficci, Pablo Funes, Greg Hornby, and Richard A. Watson. Evolutionary techniques in physical robotics. In *ICES*, pages 175–186, 2000.
- [341] Jordan B. Pollack, Hod Lipson, Gregory Hornby, and Pablo Funes. Three generations of automatically designed robots. *Artificial Life*, 7(3):215–223, 2001.
- [342] David L. Poole and Alan K. Mackworth. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2009. 2009 electronic preview / hardcover to be published 2010.
- [343] Stefan Preble, Michal Lipson, and Hod Lipson. Two-dimensional photonic crystals designed by evolutionary algorithms. *Applied Physics Letters*, 86(6):061111, 2005.

- [344] Tony J. Prescott, Peter Redgrave, and Kevin Gurney. Layered control architectures in robots and vertebrates. *Journal of Adaptive Behavior*, 7(1):99–127, 1999.
- [345] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, and James Hanan. Development models of herbaceous plants for computer imagery purposes. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 141–150, New York, NY, USA, 1988. ACM Press.
- [346] T. Quarles. SPICE3 version 3C1 users guide. Technical Report UCB/ERL M89/46, EECS Department, University of California, Berkeley, 1989.
- [347] Peter Ratiu and Ion-Florin Talos. Images in clinical medicine. The tale of Phineas Gage, digitally remastered. *New England Journal Medicine*, 351(23):e21, December 2004. Historical Article.
- [348] T.S. Ray. Aesthetically evolved virtual pets. In C. C. Maley and E. Boudreau, editors, *Proceedings of the Seventh International Conference on the Simulation and Synthesis of Living Systems (ALIFE7)*, pages 158–161. International Society for Artificial Life, The MIT Press, 2000.
- [349] Richard Reeve. *Generating walking behaviours in legged robots*. PhD thesis, University of Edinburgh, 1999.
- [350] Richard Reeve and John Halam. An analysis of neural models for walking control. *IEEE Transactions on Neural Networks*, 16(3):733–742, May 2005.
- [351] Bernard D. Reger, Karen M. Fleming, Vittorio Sanguineti, Simon Alford, and Ferdinando A. Mussa-Ivaldi. Connecting brains to robots: An artificial body for studying computational properties of neural tissues. *Artificial Life*, 6(4):307–324, 2000.
- [352] J. Reggia, M. Tagamets, J. Contreras-Vidal, D. Jacobs, S. Weems, W. Naqvi, R. Winder, T. Chabuk, J. Jung, and C. Yang. Development of a large-scale integrated neurocognitive architecture - part 1: Conceptual framework. Technical Report UMIACS-TR-2006-33, University of Maryland Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, June 2006.
- [353] J. Reggia, M. Tagamets, J. Contreras-Vidal, D. Jacobs, S. Weems, W. Naqvi, R. Winder, T. Chabuk, J. Jung, and C. Yang. Development of a large-scale integrated neurocognitive architecture - part 2: Design and architecture. Technical Report UMIACS-TR-2006-43, University of Maryland Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, October 2006.
- [354] Torsten Reil. Dynamics of gene expression in an artificial genome - implications for biological and artificial ontogeny. In *European Conference on Artificial Life*, pages 457–466, 1999.

- [355] Torsten Reil and Phil Husbands. Evolution of central pattern generators for bipedal walking in a real-time physics environment. *IEEE Transactions on Evolutionary Computation*, 6(2):159–168, April 2002.
- [356] Joseph Reisinger, Kenneth O. Stanley, and Risto Miikkulainen. Evolving reusable neural modules. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund K. Burke, Paul J. Darwen, Dipankar Dasgupta, Dario Floreano, James A. Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andrew M. Tyrrell, editors, *Genetic and Evolutionary Computation - GECCO 2004, Genetic and Evolutionary Computation Conference, Seattle, WA, USA, June 26-30, 2004, Proceedings, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 69–81. Springer, June 2004.
- [357] D Repsilber, H Liljenstrom, and SG Andersson. Reverse engineering of regulatory networks: simulation studies on a genetic algorithm approach for ranking hypotheses. *BioSystems*, 66:31–41, June 2006.
- [358] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.
- [359] John Rieffel. *Evolutionary fabrication: the co-evolution of form and formation*. PhD thesis, Brandeis University, May 2006.
- [360] John Rieffel and Jordan Pollack. Automated assembly as situated development: using artificial ontogenies to evolve buildable 3-D objects. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 99–106, Washington DC, USA, 25-29 June 2005. ACM Press.
- [361] John Rieffel and Jordan Pollack. Automated assembly as situated development: using artificial ontogenies to evolve buildable 3-D objects. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 99–106, Washington DC, USA, 25-29 June 2005. ACM Press.
- [362] F. Rieke, D. Warland, R. de R. van Steveninck, and W. Bialek. *Spikes: exploring the neural code*. The MIT Press, London, England, 1997.
- [363] M. Rohde and E. Di Paolo. t for two: Linear synergy advances the evolution of directional pointing behaviour. In M. Capcarrere, A.A. Freitas, P.J. Bentley,

- C.G. Johnson, and J. Timmis, editors, *Advances in Artificial Life: 8th European Conference, ECAL 2005, Canterbury, UK, September 5-9, 2005, Proceedings*, Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence), pages 262–271, Heidelberg, 2005. Springer.
- [364] P. Röhlfschen and E. Di Paolo. The circular topology of rhythm in asynchronous random boolean networks. *BioSystems*, 73(2):141–152, 2004.
 - [365] Raúl Rojas. *Neural networks - a systematic introduction*. Springer, 1996.
 - [366] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.
 - [367] Mike Rosenman and John Gero. Evolving designs by generating useful complex gene structures. In Peter J. Bentley, editor, *Evolutionary Design by Computers*, pages 345–364. Morgan Kaufmann, San Francisco, CA, 1999.
 - [368] D. Rumelhart and J. McClelland. *Parallel distributed processing*. MIT Press, 1986.
 - [369] Stuart Russell and Peter Norvig. *Artificial intelligence A modern approach*. Prentice-Hall, 1995.
 - [370] A. Ruvinsky and J. Sampson. *The genetics of the dog*. CABI Publishing, 2001.
 - [371] G.D. Ruxton and L.A. Saravia. The need for biological realism in the updating of cellular automata models. *Ecological Modelling*, 107:105–112, 1998.
 - [372] E. Sakamoto and H. Iba. Inferring a system of differential equations for a gene regulatory network by using genetic programming. In *Proceedings of Congress on Evolutionary Computation*, pages 720–726. IEEE Press, 2001.
 - [373] Valentina Salapura. Neural networks using bit stream arithmetic: a space efficient implementation. In *ISCAS*, pages 475–478, 1994.
 - [374] Valentina Salapura, Michael Gschwind, and Oliver Maischberger. A fast FPGA implementation of a general purpose neuron. In Reiner W. Hartenstein and Michal Servít, editors, *Field-Programmable Logic, Architectures, Synthesis and Applications, 4th International Workshop on Field-Programmable Logic and Applications, FPL '94, Prague, Czech Republic, September 7-9, 1994, Proceedings*, volume 849 of *Lecture Notes in Computer Science*, pages 175–182. Springer, 1994.
 - [375] B Samuelsson and C Troein. Superpolynomial growth in the number of attractors in Kauffman networks. *Physical Review Letters*, 90(9), March 2003.
 - [376] L. Sanchez and D. Thieffry. A logical analysis of the drosophila gap-gene system. *Journal Theoretical Biology*, 211(2):115–141, July 2001.
 - [377] M. A. Savageau. Rules for the evolution of gene circuitry. *Pacific Symposium Biocomputing*, pages 54–65, 1998.

- [378] M. Schäfer, T. Schoenauer, C. Wolff, G. Hartmann, H. Klar, and U. Ruckert. Simulation of spiking neural networks - architectures and implementations. *Neurocomputing*, 48:647–679, October 2002.
- [379] Johannes Schemmel, Andreas Gruebl, Karlheinz Meier, and Eilif Mueller. Implementing synaptic plasticity in a VLSI spiking neural network model. In *Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN 2006)*. IEEE Press, 2006.
- [380] Johannes Schemmel, Steffen Hohmann, Karlheinz Meier, and Felix Schürmann. A mixed-mode analog neural network using current-steering synapses. *Analog Integrated Circuits Signal Processing*, 38(2-3):233–244, 2004.
- [381] Johannes Schemmel, Karlheinz Meier, and Felix Schürmann. A VLSI implementation of an analog neural network suited for genetic algorithms. In *ICES '01: Proceedings of the 4th International Conference on Evolvable Systems: From Biology to Hardware*, pages 50–61, London, UK, 2001. Springer-Verlag.
- [382] B. Schonfisch and A. de Roos. Synchronous and asynchronous updating in cellular automata. *BioSystems*, 51:123–143, September 1999.
- [383] B. Schrauwen and J. Van Campenhout. BSA, a fast and accurate spike train encoding scheme. In J. Van Campenhout, editor, *Proceedings of the 2003 International Joint Conference on Neural Networks*, pages 2825–2830, Portland/OR, 7 2003. IEEE.
- [384] Benjamin Schrauwen and Jan M. Van Campenhout. Parallel hardware implementation of a broad class of spiking neurons using serial arithmetic. In *ESANN 2006, 14th European Symposium on Artificial Neural Networks, Bruges, Belg m, April 26-28, 2006, Proceedings*, pages 623–628, 2006.
- [385] Felix Schürmann, Karlheinz Meier, and Johannes Schemmel. Edge of chaos computation in mixed-mode VLSI - a hard liquid. In Lawrence K. Saul, Yair Weiss, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 1201–1208. MIT Press, Cambridge, MA, 2005.
- [386] Matthias Schwab, Martin Karrenbach, and Jon Claerbout. Making scientific computations reproducible. *Computing in Science and Engineering*, 2(6):61–67, 2000.
- [387] W B Scoville and B Milner. Loss of recent memory after bilateral hippocampal lesions. 1957. *Neuropsychiatry Clinical Neuroscience*, 12(1):103–113, Winter 2000. Biography.
- [388] Lior Segev, Ranit Aharonov, Isaac Meilijson, and Eytan Ruppin. High-dimensional analysis of evolutionary autonomous agents. *Artificial Life*, 9(1):1–20, 2003.
- [389] A. P. Shanthi, L. Karthik Singaram, and Ranjani Parthasarathi. Evolution of asynchronous sequential circuits. In *2005 NASA/DoD Conference on Evolvable*

- Hardware (EH 2005), 29 June - 1 July 2005, Washington, DC, USA*, pages 93–96. IEEE Computer Society, 2005.
- [390] Yoon-Sik Shim and Chang-Hun Kim. Generating flying creatures using body-brain co-evolution. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 276–285, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.
 - [391] Yoon-Sik Shim and Chang-Hun Kim. Evolving physically simulated flying creatures for efficient cruising. *Artificial Life*, 12(4):561–591, 2006.
 - [392] Yoon-Sik Shim, Sun Jeong Kim, and Chang-Hun Kim. Evolving flying creatures with path following behaviors. In J. Pollack, M. Bedau, P. Husbands, T. Ikegami, and R.A Watson, editors, *Artificial Life IX: Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*, page 125. International Society for Artificial Life, The MIT Press, 2004.
 - [393] Yoon-Sik Shim, Seung Yeong Shin, and Chang-Hun Kim. Two-step evolution process for path-following virtual creatures. In *17th annual conference on Computer Animation and Social Agents (CASA2004)*, July 2004.
 - [394] T.J. Shors. From stem cells to grandmother cells: how neurogenesis relates to learning and memory. *Cell Stem Cell*, 3:253–258, September 2008.
 - [395] Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449, New York, NY, USA, 1992. ACM.
 - [396] Hava T. Siegelmann and Eduardo D. Sontag. Analog computation via neural networks. *Theoretical Computer Science*, 131(2):331–360, 1994.
 - [397] Karl Sims. Evolving 3D morphology and behavior by competition. *Artificial Life*, 1(4):353–372, 1994.
 - [398] Karl Sims. Evolving virtual creatures. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 1994. ACM Press.
 - [399] Emily Singer. A wiring diagram of the brain. *MIT Technology Review*, November 2007.
 - [400] Emily Singer. The brain unmasked. *MIT Technology Review*, August 2008.
 - [401] M. Sipper and E. Ruppin. Co-evolving architectures for cellular machines. *Physica D: Nonlinear Phenomena*, 99:428–441(14), January 1997.
 - [402] M. Sipper, M. Tomassini, and M. Capcarrere. Evolving asynchronous and scalable cellular automata. In Smith et al, editor, *International Conference on Artificial Neural Networks and Genetic Algorithms'97*, pages 66–69. Springer-Verlag, June 1998.

- [403] C. Wayne Smith. *Crop production: evolution, history, and technology*. Wiley, 1995.
- [404] Russell Smith. *Open Dynamics Engine v0.5 user guide*. 2004.
- [405] T. Smith and A. Philippides. Nitric oxide signalling in real and artificial neural networks. *BT Technology Journal*, 18(4):140–149, 2000.
- [406] Lee Spector, Jon Klein, Chris Perry, and Mark Feinstein. Emergence of collective behavior in evolving populations of flying agents. *Genetic Programming and Evolvable Machines*, 6(1):111–125, 2005.
- [407] M. Spivey and R. Dale. On the continuity of mind: Toward a dynamical account of cognition. In B. Ross, editor, *The Psychology of Learning and Motivation*, volume 45. Elsevier, 2004.
- [408] Michael Spivey. *The continuity of mind*. Oxford University Press, 2006.
- [409] Kenneth O. Stanley. *Efficient evolution of neural networks through complexification*. PhD thesis, University of Texas, Austin, 2004.
- [410] Kenneth O. Stanley. Comparing artificial phenotypes with natural biological patterns. In *Genetic and Evolutionary Computation Conference (GECCO2006) Workshop Program: Complexity through Development and Self-Organizing Representations (CODESOAR)*, Seattle, WA, USA, 8-12 July 2006. ACM Press.
- [411] Kenneth O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines Special Issue on Developmental Systems*, 2007.
- [412] Kenneth O. Stanley, Ryan Cornelius, and Risto Miikkulainen. Real-time learning in the NERO video game. In R. Michael Young and John E. Laird, editors, *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference, June 1-5, 2005, Marina del Rey, California, USA*, pages 159–160. AAAI Press, 2005.
- [413] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.
- [414] Kenneth Steiglitz, Irfan Kamal, and Arthur Watson. Embedding computation in one-dimensional automata by phase coding solitons. *IEEE Transactions on Computers*, 37(2):138–145, 1988.
- [415] A. Steinlage and G. Schoener. Dynamical systems for the behavioral organization of autonomous robot navigation. In P. S. Schenker and G. T. McKee, editors, *Proceedings of the SPIE — Sensor Fusion and Decentralized Control in Robotic Systems*, volume 3523 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, pages 169–180, October 1998.
- [416] S. Still and M. W. Tilden. Controller for a four legged walking machine. *Neuromorphic Systems: Engineering silicon from neurobiology*, 1998.

- [417] Adrian Stoica, Ricardo Zebulum, and Didier Keymeulen. Progress and challenges in building evolvable devices. In *EH '01: Proceedings of the The 3rd NASA/DoD Workshop on Evolvable Hardware*, page 33, Washington, DC, USA, 2001. IEEE Computer Society.
- [418] Emanuele Stomeo, Tatiana Kalganova, Cyrille Lambert, N. Lipnitsakya, and Y. Yatskevich. On evolution of relatively large combinational logic circuits. In *2005 NASA / DoD Conference on Evolvable Hardware (EH 2005), 29 June - 1 July 2005, Washington, DC, USA*, pages 59–66. IEEE Computer Society, 2005.
- [419] T. Streeter, J. Oliver, and A. Sannier. Verve: A general purpose open source reinforcement learning toolkit. In *Proceedings of the ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2006.
- [420] Tyler Edward Streeter. Design and implementation of general purpose reinforcement learning agents. Master's thesis, Iowa State University, 2003.
- [421] Masahiro Sugimoto, Kouichi Takahashi, Tomoya Kitayama, Daiki Ito, and Masaru Tomita. Distributed cell biology simulations with e-cell system. pages 20–31. 2005.
- [422] John D. Sutherland, Matthew W. Powner, and Beatrie Gerland. Synthesis of activated pyrimidine ribonucleotides in prebiotically plausible conditions. *Nature*, 459(7244):239–242, May 2009.
- [423] T. Suzudo. Spatial pattern formation in asynchronous cellular automata with mass conservation. *Physica A: Statistical Mechanics and its Applications*, 343:185–200, November 2004.
- [424] Tomoaki Suzudo. Searching for pattern-forming asynchronous cellular automata - an evolutionary approach. In Peter M. A. Sloot, Bastien Chopard, and Alfons G. Hoekstra, editors, *Cellular Automata, 6th International Conference on Cellular Automata for Research and Industry, ACRI 2004, Amsterdam, The Netherlands, October 25-28, 2004, Proceedings*, volume 3305 of *Lecture Notes in Computer Science*, pages 151–160. Springer, 2004.
- [425] Michael Swaine. AI: It's OK again! *Dr. Dobb's Journal*, September 2007.
- [426] N.I. Syed, A.G. Bulloch, and K Lukowiak. In vitro reconstruction of the respiratory central pattern generator of the mollusk lymnaea. *Science*, pages 282–285, October 1990.
- [427] G. Taga. A model of the neuro-musculo-skeletal system for human locomotion. 1. emergence of basic gait. *Biological Cybernetics*, 73(2):97–111, 1995.
- [428] G Taga, Y Yamaguchi, and H Shimizu. Self-organized control of bipedal locomotion by neural oscillators in unpredictable environment. *Biological Cybernetics*, 65(3):147–159, 1991.

- [429] Kay Chen Tan, Tong Heng Lee, and Eik Fun Khor. Evolutionary algorithms with dynamic population size and local exploration for multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 5(6):565–588, 2001.
- [430] Ivan Tanev and Thomas Ray. Evolution of sidewinding locomotion of simulated limbless, wheelless robots. In M. Sugisaka and H. Tanaka, editors, *Proceedings of the 9th International Symposium on Artificial Life and Robotics (AROB-04)*, volume 2, pages 472–475, 2004.
- [431] Ivan Tanev and Thomas S. Ray. Evolution of sidewinding locomotion of simulated limbless, wheelless robots. *Artificial Life and Robotics*, 9(3):117–122, 2005.
- [432] C.Z. Tang and H.K. Kwan. Digital implementation of neural networks with quantized neurons. In *Proceedings of 1997 IEEE International Symposium on Circuits and Systems*, volume 1 of *ISCAS*, pages 649–652, June 1997.
- [433] Toshiharu Taura and Ichiro Nagasaka. Adaptive-growth-type 3D representation for configuration design. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing (AI EDAM)*, 13(3):171–184, 1999.
- [434] R Tawel, N Aranki, G.V. Pusko, K.A. Marko, L.A. Feldkamp, James J.V., G Jesion, and T.M. Feldkamp. Custom VLSI ASIC for automotive applications with recurrent networks. In IEEE Neural Networks Council, editor, *Proceedings of the International Joint Conference on Neural Networks (IJCNN'98)*, 1998.
- [435] Tim Taylor. Artificial life techniques for generating controllers for physically modelled characters. In Quasim H. Mehdi and Norman E. Gough, editors, *1st International Conference on Intelligent Games and Simulation (GAME-ON 2000), 11-12 November 2000, London, UK*, pages 89–95, 2000.
- [436] Tim Taylor and Colm Massey. Recent developments in the evolution of morphologies and controllers for physically simulated creatures. *Journal of Artificial Life*, 7(1):77–87, 2000.
- [437] D. Thierens. Adaptive mutation rate control schemes in genetic algorithms. In *Proceedings of IEEE Congress on Evolutionary Computation (CEC 2002)*, volume 1, pages 980–985, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
- [438] Dale Thomas. Aesthetic selection of morphogenetic art forms. *Kybernetes*, 32(1-2):144–155, 2002.
- [439] Ray S. Thomas, D Thieffry, and M Kaufman. Dynamical behaviour of biological regulatory networks. *Bulletin of Mathematical Biology*, 57:247–276, March 1995.
- [440] A. Thompson. *Hardware evolution: automatic design of electronic circuits in reconfigurable hardware by artificial evolution*. Distinguished dissertation series. Springer-Verlag, 1998.

- [441] Adrian Thompson. Evolving electronic robot controller that exploit hardware resources. In *European Conference on Artificial Life*, pages 640–656, 1995.
- [442] Adrian Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In Tetsuya Higuchi, Masaya Iwata, and Weixin Liu, editors, *Evolvable Systems: From Biology to Hardware, First International Conference, ICES 96, Tsukuba, Japan, October 7-8, 1996, Proceedings*, volume 1259 of *Lecture Notes in Computer Science*, pages 390–405. Springer, 1996.
- [443] Adrian Thompson. Silicon evolution. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 444–452, Stanford University, CA, USA, 1996. MIT Press.
- [444] Adrian Thompson, Paul J. Layzell, and Ricardo Salem Zebulum. Explorations in design space: unconventional electronics design through artificial evolution. *IEEE Transactions on Evolutionary Computation*, 3(3):167–196, 1999.
- [445] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: a new environment for modeling*. MIT Press, Cambridge, MA, USA, 1987.
- [446] Julian Togelius. Evolution of a subsumption architecture neurocontroller. *Journal of Intelligent & Fuzzy Systems: Applications in Engineering and Technology*, 15(1):15–20, 2004.
- [447] Seiji Tokura, Akio Ishiguro, and Shigeru Okuma. Hardware implementation of neuromodulated neural network for a CPU-less autonomous mobile robot. *Advanced Robotics*, 20(12):1341–1358, 2006.
- [448] S. Tomasula. Genetic art and the aesthetics of biology. *Leonardo*, 35(2):137–144, April 2002.
- [449] Martin Trefzer, Jörg Langeheine, Karlheinz Meier, and Johannes Schemmel. A modular framework for the evolution of circuits on configurable transistor array architectures. In Adrian Stoica, Tughrul Arslan, Martin Suess, Senay Yalçın, Didier Keymeulen, Tetsuya Higuchi, Ricardo Salem Zebulum, and Nizamettin Aydin, editors, *First NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2006), 15-18 June 2006, Istanbul, Turkey*, pages 32–42. IEEE Computer Society, 2006.
- [450] Vito Trianni, Roderich Groß, Thomas Halva Labella, Erol Sahin, and Marco Dorigo. Evolving aggregation behaviors in a swarm of robots. In Wolfgang Banzhaf, Thomas Christaller, Peter Dittrich, Jan T. Kim, and Jens Ziegler, editors, *Advances in Artificial Life, 7th European Conference, ECAL 2003, Dortmund, Germany, September 14-17, 2003, Proceedings*, volume 2801 of *Lecture Notes in Computer Science*, pages 865–874. Springer, 2003.
- [451] V. N. Tsytovich, G. E. Morfill, V. E. Fortov, N. G. Gusein-Zade, B. A. Klumov, and S. V. Vladimirov. From plasma crystals and helical structures towards inorganic living matter. *New Journal of Physics*, 9(8):263, 2007.

- [452] L. Turin. A spectroscopic mechanism for primary olfactory reception. *Chemical Senses*, 21(773-791), 1996.
- [453] Andrew Twyman. The threats of distributed cracking. Technical report, MIT 6.806/STS085: Ethics and law on the electronic frontier, Massachusetts, 1997.
- [454] Obinwanne Ugwonali. The role of white yams in the increased incidence of multiple births in southwestern nigeria. Master's thesis, Yale Medicine, 1999.
- [455] Andres Upegui, Carlos Andrés Peña-Reyes, and Eduardo Sanchez. A functional spiking neuron hardware oriented model. In José Mira and José R. Álvarez, editors, *Artificial Neural Nets Problem Solving Methods, 7th International Work-Conference on Artificial and Natural Neural Networks, IWANN2003, Maó, Menorca, Spain, June 3-6, 2003 Proceedings, Part I*, volume 2686 of *Lecture Notes in Computer Science*, pages 136–143. Springer, 2003.
- [456] Andres Upegui, Carlos Andrés Peña-Reyes, and Eduardo Sanchez. An FPGA platform for on-line topology exploration of spiking neural networks. *Microprocessors and Microsystems*, 29(5):211–223, 2005.
- [457] Floris van Breugel and Hod Lipson. Evolving buildable flapping ornithopters. In *Late Breaking Papers at the GECCO 2005 Genetic and evolutionary computation conference*, Washington DC, USA, 25-29 June 2005. ACM Press.
- [458] Patrick Vandewalle, Jelena Kovacevic, and Martin Vetterli. What, why and how of reproducible research in signal processing. *IEEE Signal Processing Magazine*, June 2008.
- [459] Ajit Varki and Tasha K. Altheide. Comparing the human and chimpanzee genomes: Searching for needles in a haystack. *Genome Research*, 15(12):1746–1758, December 2005.
- [460] V. Vassilev, D. Job, and J. Miller. Towards the automatic design of more efficient digital circuits. In Jason Lohn, Adrian Stoica, and Didier Keymeulen, editors, *The Second NASA/DoD workshop on Evolvable Hardware*, pages 151–160, Palo Alto, California, 2000. IEEE Computer Society.
- [461] E. Vaughan, E. A. Di Paolo, and I. Harvey. The evolution of control and adaptation in a 3D powered passive dynamic walker. In J. Pollack, M. Bedau, P. Husbands, T. Ikegami, and R.A Watson, editors, *Artificial Life IX: Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*, pages 139–145. International Society for Artificial Life, The MIT Press, 2004.
- [462] E. Vaughan, E. A. Di Paolo, and I. Harvey. The tango of a load balancing biped. In *Proceedings of the Seventh International Conference on Climbing and Walking Robots (CLAWAR)*, 2004.
- [463] Eric Vaughan. Evolution of 3D bipedal walking with hips and ankles. Master's thesis, Evolutionary and adaptive systems group, University of Sussex, 2003.

- [464] Ganesh Venkataraman, Sudhakar M. Reddy, and Irith Pomeranz. GALLOP: Genetic algorithm based low power FSM synthesis by simultaneous partitioning and state assignment. In *Proceedings of the 16th International Conference on VLSI Design*, page 533, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
- [465] J. Ventrella. Disney meets darwin - the evolution of funny animated figures. In *CA '95: Proceedings of the Computer Animation*, page 35, Los Alamitos, CA, USA, 1995. IEEE Computer Society.
- [466] Jeffrey Ventrella. Genepool: Exploring the interaction between natural selection and sexual selection. In Andrew Adamatzky and Maciej Komosinski, editors, *Artificial Life Models in Software*, chapter 4. Springer-Verlag, New York, 2005.
- [467] Benjamin F F. Voight, Sridhar Kudaravalli, Xiaoquan Wen, and Jonathan K K. Pritchard. A map of recent positive selection in the human genome. *Public Library of Science (PLoS) Biology*, 4(3), March 2006.
- [468] Bartelemy von Haller. Neubot project: framework for simulating modular robots and self-organisation of locomotion under water. Master's thesis, Swiss Federal Institute of Technology (EPFL) Lausanne, Lausanne, 2005.
- [469] Bartelemy von Haller, A. Ijspeert, and D. Floreano. Co-evolution of structures and controllers for Neubot underwater modular robots. In M. Capcarrere, A.A. Freitas, P.J. Bentley, C.G. Johnson, and J. Timmis, editors, *Advances in Artificial Life: 8th European Conference, ECAL 2005, Canterbury, UK, September 5-9, 2005, Proceedings*, Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence), Heidelberg, 2005. Springer.
- [470] Henning U Voss, Aziz M Uluc, Jonathan P Dyke, Richard Watts, Erik J Kobylarz, Bruce D McCandliss, Linda A Heier, Bradley J Beattie, Klaus A Hamacher, Shankar Vallabhajosula, Stanley J Goldsmith, Douglas Ballon, Joseph T Giacino, and Nicholas D Schiff. Possible axonal regrowth in late recovery from the minimally conscious state. *Clinical Investigation*, 116(7):2005–2011, July 2006.
- [471] Jilles Vreeken. Spiking neural networks, an introduction. Technical Report UU-CS-2003-008, Institute of Information and Computing Sciences, Utrecht University, 2003.
- [472] C. H. Waddington. Canalization of development and the inheritance of acquired characters. In *Adaptive individuals in evolving populations: models and algorithms*, pages 91–97. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
- [473] Sayaka Wakayama, Yumi Kawahara, Chong Li, Kazuo Yamagata, Louis Yuge, and Teruhiko Wakayama. Detrimental effects of microgravity on mouse preimplantation development in vitro. *PLoS ONE*, 4(8):e6753, 08 2009.

- [474] Kevin Warwick. *March of the machines: the breakthrough in artificial intelligence*. University of Illinois Press, 2004.
- [475] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.
- [476] Justin Werfel, Melanie Mitchell, and James P. Crutchfield. Resource sharing and coevolution in evolving cellular automata. *IEEE Transactions on Evolutionary Computation*, 4(4):388–393, 2000.
- [477] R.E. Wheeler. Algdesign. *The R project for statistical computing*, 2004.
- [478] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *Neurocomputing: foundations of research*, pages 123–134. MIT Press, Cambridge, MA, USA, 1988.
- [479] Alexis P. Wieland. Evolving neural network controllers for unstable systems. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'91)*, volume II, pages 667–673, 1991.
- [480] Stephen Wolfram. *A new kind of science*. Wolfram Media Inc., 2002.
- [481] A. Wuensche. Discrete dynamical networks and their attractor basins. In Russell Standish, Bruce Henry, Simon Watt, Robert Marks, Robert Stocker, David Green, Steve Keen, and Terry Bossomaier, editors, *Proceedings of Complex Systems '98*, University of New South Wales, Sydney, Australia., 1998.
- [482] A. Wuensche. Discrete dynamics lab: Tools for investigating cellular automata and discrete dynamical networks. In Andrew Adamatzky and Maciej Komosinski, editors, *Artificial Life Models in Software*, chapter 11, pages 263–297. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [483] Juan M. Xicotencatl and Miguel Arias-Estrada. FPGA based high density spiking neural network array. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings*, volume 2778 of *Lecture Notes in Computer Science*, pages 1053–1056, Berlin, Heidelberg, September 2003. Springer Verlag.
- [484] S. Xu, S. K. Talwar, E. S. Hawley, L. Li, and J. K. Chapin. A multi-channel telemetry system for brain microstimulation in freely roaming animals. *Journal of Neuroscience Methods*, 133(1-2):57–63, February 2004.
- [485] Fuminori Yamasaki, Tatsuya Matsui, Takahiro Miyashita, and Hiroaki Kitano. Pino the humanoid: A basic architecture. In *RoboCup 2000: Robot Soccer World Cup IV*, pages 269–279, London, UK, 2001. Springer-Verlag.
- [486] X. Yao. A review of evolutionary artificial neural networks. *International Journal of Intelligent Systems*, 8(4):203–222, 1993.

- [487] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.
- [488] Juan Cristóbal Zagal and Javier Ruiz del Solar. Uchilsim: A dynamically and visually realistic simulator for the robocup four legged league. In Daniele Nardi, Martin Riedmiller, Claude Sammut, and José Santos-Victor, editors, *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276 of *Lecture Notes in Computer Science*, pages 34–45. Springer, 2004.
- [489] Franco Zambonelli, Marco Mamei, and Andrea Roli. What can cellular automata tell us about the behavior of large multi-agent systems? In Alessandro F. Garcia, Carlos José Pereira de Lucena, Franco Zambonelli, Andrea Omicini, and Jaelson Castro, editors, *SELMAS*, volume 2603 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2002.
- [490] G Zeck and P Fromherz. Noninvasive neuroelectronic interfacing with synaptically connected snail neurons immobilized on a semiconductor chip. *Proceedings of the National Academy of Sciences of the United States of America*, 98(18):10457–10462, August 2001.
- [491] Andreas Zell, Niels Mache, Ralf Huebner, Michael Schmalzl, Tilman Sommer, and Thomas Korb. SNNS: Stuttgart neural network simulator. Technical report, University of Stuttgart, Stuttgart, 1992.
- [492] J. Zhu and P. Sutton. FPGA implementations of neural networks - a survey of a decade of progress. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings*, volume 2778 of *Lecture Notes in Computer Science*, pages 1062–1066, Berlin, Heidelberg, September 2003. Springer Verlag.
- [493] J.C. Zufferey, A. Guanella, A. Beyeler, and D. Floreano. Flying over the reality gap: from simulated to real indoor airships. *Autonomous Robots*, 21(3):243–254, 2006.
- [494] Jean-Christophe Zufferey. *Bio-inspired vision-based flying robots*. PhD thesis, Swiss Federal Institute of Technology (EPFL) Lausanne, Lausanne, 2005. Prix Asea Brown Boveri Ltd (ABB) (2006).
- [495] Jean-Christophe Zufferey, Dario Floreano, Matthijs van Leeuwen, and Tancredi Merenda. Evolving vision-based flying robots. In Heinrich H. Bülfhoff, Seong-Whan Lee, Tomaso Poggio, and Christian Wallraven, editors, *Biologically Motivated Computer Vision Second International Workshop, BMVC 2002, Tübingen, Germany, November 22-24, 2002, Proceedings*, volume 2525 of *Lecture Notes in Computer Science*, pages 592–600. Springer, 2002.
- [496] Viktor Zykov, Josh Bongard, and Hod Lipson. Evolving dynamic gaits on a physical robot. In Maarten Keijzer, editor, *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference*, Seattle, Washington, USA, 26 July 2004. Springer.