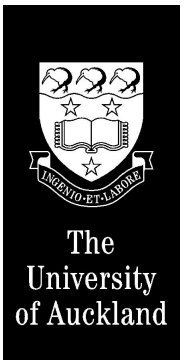
INTELLIGENT MOTION CONTROL WITH AN ARTIFICIAL CEREBELLUM

Russell L. Smith
July 1998

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN ENGINEERING.



THE DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING,
UNIVERSITY OF AUCKLAND,
NEW ZEALAND

The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with eager feet,
Until it joins some larger way
Where many paths and errands meet.
And whither then? I cannot say.

—J.R.R. Tolkien

To my friends and lab-mates —
Aaron, Brian, Chris, Dave,
Dominic, Gavin, Geoff, Lee,
Melissa, Russ M., Steve, Sylvia,
Tony and Woei . . .
. . . It's been a long journey,
but thanks to you, a fun one.

To my family, Mum, Dad and Raewyn.

To my supervisor, George.

Abstract

This thesis describes a novel approach for adaptive optimal control and demonstrates its application to a variety of systems, including motion control learning for legged robots. The new controller, called "FOX", uses a modified form of Albus's CMAC neural network. It is trained to generate control signals that minimize a system's performance error. A theoretical consideration of the adaptive control problem is used to show that FOX must assign each CMAC weight an "eligibility" value which controls how that weight is updated. FOX thus implements a kind of reinforcement learning which makes it functionally similar to the cerebellum (a part of the brain that modulates movement). A highly efficient implementation is described which makes FOX suitable for on-line control.

FOX requires a small amount of dynamical information about the system being controlled: the system's impulse response is used to choose the rules that update the eligibility values. A FOX-based controller design methodology is developed, and FOX is tested on four control problems: controlling a simulated linear system, controlling a model gantry crane, balancing an inverted pendulum on a cart, and making a wheeled robot follow a path. In each case FOX is effective: it associates sensor values with (and anticipates) the correct control actions, it compensates for system nonlinearities, and it provides robust control as long as the training is comprehensive enough.

FOX is also applied to the control of a simulated hopping monoped, and a walking biped. FOX learns parameters that fine tune the movements of pre-programmed controllers, in a manner analogous to the cerebellar modulation of spinal cord reflexes in human movement. The robots are successfully taught how to move with a steady gait along flat ground, in any direction, and how to climb and descend slopes.

Contents

[1] This appendix is derived from the author's paper in the Proceedings of the 1995 ANNES conference [114].

Nomenclature

Acronyms

CMAC	**C**erebellar **m**odel **a**rticulation **c**ontroller.
FBE	**F**eed**b**ack **e**rror.
FOX	**F**airly **o**bvious e**x**tension (to the CMAC).
IFC	**I**nternal **f**eedback **c**ontroller.
WIB	**W**eight **i**ndex **b**uffer.

CMAC symbols (introduced in Chapter 3)

y_i	Inputs.
x_i	Outputs.
z_{ij}	Feature detecting neurons (2-layer example).
a_{ij}	Association neurons (2-layer example).
n_y	Number of inputs y_i.
n_x	Number of outputs x_i.
n_a	Number of association neurons activated per input, or the number of AU (association unit) tables.
n_v	Number of association neurons a_{ij}.
n_w	Total number of physical weights per output.
w_{ijk}	Weights (neural network model).
d_j^i	Displacement for AU i along input y_j.
q_i	Quantized version of input y_i.
p_j^i	Table index of AU i along axis y_j.
α	Learning rate.
e_i	Error signal for output x_i.
f_v	Virtual address generating function.
f_h	Hashing function.

n_p	Number of physical weights.
$\min_i$, $\max_i$	Minimum and maximum values for y_i.
res_i	Input resolution; the number of quantization steps for input y_i.
μ_i	Index into the weight tables for AU i.
$W_j[i]$	Weight i in the weight table for output x_j.

FBE symbols (introduced in Chapter 4)

$y(t)$	System state.
$x(t)$	Control force (controller output, system input).
$e(t)$	Error signal.
$y_d(t)$	Desired system state over time.
$F(y, x)$	Controlled system function.
A	Trainable controller.
Q	Fixed feedback controller.
x_a	Output of A.
x_q	Output of Q.
w	Weights which parameterize A.
p	Position of the simple 2nd order example system.
p_d	Desired value of p.

FOX symbols (introduced in Chapter 5 and Chapter 6)

$F(\mathbf{y}_i, \mathbf{x}_i)$	The system function, which takes the current state and control input and generates the next time step's state.
F^*	A linear approximation to F.
$C(\mathbf{y}_i)$	The "critic" function, which generates a scalar error value at each time step.
$\mathbf{y}_i$	The system state vector for time i. Size $n_y \times 1$.
$\mathbf{x}_i$	The system control input vector for time i (the output of the CMAC controller). Size $n_x \times 1$.
$\mathbf{s}_i$	A vector of CMAC association-unit "sensors" that encodes the current system state. Size $n_s \times 1$.
W	The matrix of CMAC weights. Note that $\mathbf{x}_i = W\,\mathbf{s}_i$. Size $n_x \times n_s$.
e_i	A scalar error value generated by the critic function for time i.
y_i^d	A scalar position reference (for time i) used in the calculation of e.

E	The total scalar error of the entire system.
w_{pq}	Element (p,q) of the matrix W.
ξ_i^{pq}	The eligibility vector for weight w_{pq}. Size $n_y \times 1$.
$C\xi$	The eligibility profile (scalar).
$\hat{\mathbf{s}}_i^{pq}$	A synonym for $\mathrm{d}\mathbf{x}_i/\mathrm{d}w_{pq}$. $\hat{\mathbf{s}}_i^{pq}$ is all zero except for its p'th element which is equal to the q'th element of $\mathbf{s}_i$. Size $n_x \times 1$.
n_w	Total number of weights, $n_w = n_s\, n_x$.
n_a	Number of association units in the CMAC, i.e. the number of nonzero elements in $\mathbf{s}_i$.
A	A system matrix for the linear system F^*. Size $n_y \times n_y$.
B	A system matrix for the linear system F^*. Size $n_y \times n_x$.
C	The matrix for the simple linear critic function. Size $1 \times n_y$.

FOX learning rate symbols

α	The main FOX learning rate (scalar).
β	The output limiting FOX learning rate (scalar).
γ	The output derivative limiting FOX learning rate (scalar).
α_1, α_2	The overshoot error function FOX learning rates (scalars).

Eligibility profile cookbook symbols

k_a, k_b	Parameters for the selection of over-damped second order eligibility profiles (see Appendix E).
$t$max	Parameter for selecting critically damped second order eligibility profiles (see Appendix E).

FOX algorithm symbols

σ	Cut-off (decay-to-zero) point for the eligibility profile.
δ	WIB buffer size, $\delta = \sigma + 2$
λ_i	The WIB buffer, an array of δ vectors each of size $1 \times n_y$.
T_t	The current time step.
g	The position of the current time step in the WIB.
T_0	The time step at λ_0.
Λ	The accumulated value of: $e\, CA^i$ (size $1 \times n_y$).
Γ	The current value of: CA^g (size $1 \times n_y$).

Contents of the CDROM

This thesis is accompanied by a CDROM containing several movies in MPEG and AVI format. The subdirectory `thesis` on the CDROM also contains postscript files for this thesis (individual chapters as well as the entire document).

Inverted Pendulum Movie (AVI)

- `inverted.avi` : This shows various stages of the inverted pendulum experiment:
  - PD controller with a zero reference.
  - Trained FOX controller with a zero reference.
  - Trained FOX controller with a square wave reference.
  - PD controller with reference $= \sin(x) + \cos(2x)$.
  - Trained FOX controller with reference $= \sin(x) + \cos(2x)$.

Hopping Robot Movies (MPEG)

- `hopper1.mpg` : This shows four out-takes from the training of the hopping robot. Watch it fall over in various ways as it learns.
- `hopper2.mpg` : This shows the fully trained hopping robot following a path through its environment. It can go up and down a ramp without falling over.
- `hopper3.mpg` : This shows one of the hopping robot's failed attempts to climb the ramp. A foot trail is rendered, which can help diagnose the problem.

Biped Walking Movies (MPEG)

- `walk1.mpg` : Only the motion sequencer is used in the controller, no other controller modules are active. The biped makes stereotyped stepping movements and falls down immediately.
- `walk2.mpg` : The effect of hip side compensation is demonstrated. Before hip side compensation training the biped sways from side to side as it walks. After training the biped stays more upright.
- `walk3.mpg` : The combined effect of hip twist compensation and arm swing is demonstrated. Before these things are trained the biped twists from left to right as it walks. After training the biped is steadier.

- `walk4.mpg` : The effect of global side drift compensation is demonstrated. Before compensation training the biped becomes unstable when it leaves its desired path.
- `walk5.mpg` : A tripping event is demonstrated — the biped's foot touches the ground prematurely during a walking cycle.
- `walk6.mpg` : The effect of leg placement is demonstrated. Without it the biped makes no attempt to correct a falling motion. With correct leg placement the biped will place the feet to try and keep the body upright.
- `walk8.mpg` : Some out-takes from training are shown — the biped falls over in various ways as it learns to walk.
- `walkt1.mpg`, `walkt2.mpg`, `walkt3.mpg` : Successful walking — walking with a steady gait along flat ground, walking at various speeds, changing direction, and walking up and down slopes. Three copies of the movie are provided which use different MPEG bit rates (there are differences in movie quality).
- `explode.mpg` : This shows what can happen when the simulation goes astray: an incorrectly configured joint controller produces a large force that sends the robot flying into the air.

Chapter 1

Introduction

1.1 Intelligent motion control

All animals, from insects to people, are endowed with a sophisticated range of behaviors that allow them to interact appropriately with their world. They are capable of complex sensory processing, versatile execution of coordinated movements, and (in higher animals) planning of actions that will accomplish some task. Their behavior is *motivated* or goal driven rather than just being purely reactive. Their behavior is *adaptive*, in the sense that it can be automatically adjusted to suit unanticipated variations in their unstructured environments. And most importantly (in higher animals), those skills that they were not born with they can *learn*. The intelligent control problem is to design robots (or in general, automatons) that have some of these properties.

The problem of intelligent control is wide ranging, involving many disciplines of science and engineering. This thesis is concerned with only one aspect: the *motion control problem*—that is, how to make a robot move in a dynamic and coordinated way within its environment. Of course a useful robot will also need sensory processing and planning capabilities, but these problems will not be considered at all here.

In the emerging science of autonomous robots, the motion control problem is often ignored. This is because most existing robots use wheels for locomotion, so the problem is reduced to achieving a desired robot direction and forward speed, which is easily solved with current control technology. But the dynamical aspects of intelligent control are becoming more important. Consider the problems which would benefit from intelligent motion control:

- Controlling a wheeled autonomous robot where the mass-to-desired-speed ratio is high enough that the robot's dynamics play an important part in its trajectory, so that simple position-reference control strategies can not be used.
- Controlling a legged robot to achieve stable locomotion, where the desired leg motions are not known.
- Controlling a mechanically imperfect robot arm to achieve high speed placement of the end-effector. Mechanical imperfections such as flexibility in the links and slip in the drive-train are hard to deal with using conventional motion controller hardware.
- In general, the control of an arbitrary nonlinear mechanical system (such as an inverted pendulum on a cart) so that its performance is optimized in some manner.

Learning is an essential component of intelligent control. Consider that a non-learning controller must be carefully constructed to suit the environment: either a good system model must be constructed and a controller derived analytically from it, or a controller must be built with a number of unknown parameters which are fine tuned by hand as the system operates. Both approaches are very time consuming for all but the simplest systems. In learning-control the unknown controller parameters are automatically adjusted on-line as the system operates. This has the added advantage that the controller can compensate for unanticipated changes in the controlled system or environment.

The problem of learning motion control is encompassed by the highly developed field of *adaptive control* [51]. The two fields have similar goals but differ in scope and execution. A traditional adaptive controller tends to adjust a small number of parameters for linear systems to achieve optimal performance. Typically a learning process adjusts the parameters slowly through some least squares technique. Adaptive control can be highly robust, so there are many industrial applications. In contrast, a typical "intelligent" motion controller tends to output the control signals directly from a neural network (or similar device). A much larger number of parameters are adjusted by learning. These techniques are much more experimental and much less robust, although they have the potential to outperform more traditional approaches.

In this thesis a novel biologically motivated learning-control method called the "FOX" algorithm is developed and tested. It is shown that FOX is useful by itself for adaptive nonlinear optimal control. A "toolbox" approach to intelligent controller design is advocated, and it is shown that FOX is useful as a generic *learning component* in intelligent controllers.

1.2 Thesis overview

The remainder of this chapter gives some general background on autonomous robots and learning. The next three chapters look at three separate aspects of intelligent motion control. Chapter two gives some background information on how biological motor control is achieved in humans and other animals. This is important for two reasons: first, the FOX algorithm is a model of the cerebellum, a part of the brain that modulates movement. Second, one of the organizing principles in the brain is that neural circuits in the brain stem modulate reflex-implementing circuits in the spinal cord—a principle that is useful when applied to the design of robot controllers.

Chapter three looks at the CMAC neural network on which FOX is based. This chapter is largely a tutorial, although the performance of the CMAC is analyzed in some detail.

Chapter four introduces a control method called "feedback-error" (FBE). FBE was originally conceived by Kawato [60] as a model of human motor control, but in the form presented here it is simply an adaptive control methodology. FBE is important because it is a starting point and useful comparison for the FOX controller.

In chapter five the theory of the FOX controller is explained. A theoretical consideration of the adaptive control problem (using a CMAC) is used to derive the FOX algorithm. Aspects of the three previous chapters are pulled together to show how FOX overcomes the deficiencies of FBE, and how it is a model of the cerebellum. An efficient implementation of the FOX algorithm is described that achieves high speed by overcoming several practical problems.

In chapter six, FOX is put to the test. First a design methodology for FOX-based systems is described. Then FOX is successfully tested on four problems: controlling a simulated linear system, controlling a model gantry crane, balancing an inverted pendulum on a cart, and making a wheeled robot follow a path. In the process various design issues are explored.

In chapter seven it is shown that FOX can be a useful component in the controllers for more complex

robots: a simulated hopping monoped, and a simulated walking biped. FOX modules in these robots learn various parameters that fine tune the movements of their hard-wired controllers, in a manner analogous to the cerebellar modulation of spinal cord reflexes in human movement. Note that the CDROM that accompanies this thesis contains several MPEG and AVI format movies that have been created to illustrate various aspects of robot motion and training.

Finally, in chapter eight a list of unsolved problems and possible future work will be presented, and the main conclusions of the thesis will be summarized.

1.3 Thesis contribution

It will be seen later that the FOX controller adds an eligibility value to each CMAC weight. This by itself is not a new concept [67], but FOX is unique in several respects:

- FOX's eligibility values can be vectors, not just scalars. Vector eligibility values have not been used in any existing eligibility-based reinforcement learning techniques, to the author's knowledge.
- It is shown that the best eligibility update equations can be derived from easily measured dynamics of the controlled system. This allows a practical FOX design methodology to be created. Previous studies [68] have only provided loose guidelines for updating scalar eligibilities.
- A highly efficient FOX implementation is given which overcomes the need to update each weight's eligibility at each time step. This is more sophisticated, faster and more useful than previous approaches such as [48].
- FOX has learned to control the classically "hard" inverted pendulum system. Many other approaches have also solved this problem, but the FOX solution is particularly elegant.
- A simulated biped robot is taught to successfully walk—this is not new, as others have demonstrated successful biped locomotion. The novel aspect is the controller architecture that was used, where FOX modules were used as learning components.

1.4 Background on robots and learning

Before describing the CMAC and the FOX algorithm some general background information on learning methods and autonomous robot design will be presented. Some basic material on neural networks, training and generalization is given in Appendix A. Virtually every autonomous robot designer uses a different combination of controller paradigm and learning method. As a result the field of autonomous robots is very broad, but shallow, with few unifying threads. Many working systems have been demonstrated but no "killer paradigm" has been found. The many current techniques used are drawn from so many areas that it is sometimes difficult to make meaningful comparisons between different systems.

1.4.1 Why is intelligent control useful?

Intelligent control is necessary for autonomous robots which must survive in unstructured environments without continuous human guidance. Examples include robot sentries, intra-office delivery robots, and robot tour guides. Unlike the typical factory robot they may have to negotiate environments which are complex, changeable, full of obstacles and possibly hostile.

Intelligent control is also necessary when there is a time lag between operator commands and robot execution. A example is NASA's six-wheeled robotic rover, Sojourner, that was placed on the surface of Mars in July 1997 as part of the Mars Pathfinder project [74]. The rover had to be partly autonomous because of the large communication delay between Earth and Mars[1]. The rover navigated between waypoints specified by an earth bound operator, avoiding obstacles along the way.

Intelligent control is useful in telepresence applications, where a human operator issues high-level commands from a remote location and the robot complies using its own behavioral resources to implement the lower-level subtasks that are required.

Finally, a relatively new and speculative application of intelligent control is to control virtual actors. For example, in some situations an animator would prefer to be able to ask a virtual actor to "walk into the room and sit down" rather than specifying the detailed motions required.

1.4.2 Modular behavior and the break with classical AI

The classical artificial intelligence (AI) approach to controlling autonomous systems is to break the problem up in to functional modules such as sensory perception, environmental modeling, planning of actions and execution of those plans [19]. Various techniques from the AI toolbox are used within each module, such as knowledge representation schemes, goal directed searching and reasoning [121]. However, it has been argued that classical AI can not produce systems that are robust in real-world environments [13]. AI "intelligence" is notoriously narrow and brittle and is highly sensitive to the representation schemes used. AI knowledge is encapsulated in symbolic abstractions, and therefore is inflexible to deviations in the real world. AI systems do not seem to have much "common sense".

An alternative paradigm for the design of intelligent controllers is that of "modular behavior". The idea is that the controller contains modules which each perform some simple task oriented function. The overall behavior of the robot emerges from the interaction of these modules, rather than being specified explicitly.

For example, Brooks' subsumption architecture [19, 20] consists of modules containing state machines and timers that each implement some simple behavior. State machine outputs can be combined so that the output from one suppresses that of another. In this way higher level modules take over when the lower levels are inadequate to the task. The subsumption architecture has been used to make robust controllers for wheeled and legged robots, though the networks have to be carefully constructed to get the desired behavior.

Another example is Beer's artificial insect [13, 15, 14]. It is a hexapod robot that is controlled by an artificial nervous system made up of about 80 biologically realistic neurons. The neural network is designed rather than trained: it contains groups of neurons dedicated to particular tasks (such as leg control), which interact to achieve overall coordinated movement. The robot is capable of walking with various gaits and performing simple tasks such as wall following and food finding.

These two examples, though different in implementation are similar in principle: they are designed from the bottom up, by adding units that interact with the existing structure to create new behavior. Such systems are constructed especially to survive in their environments. They are not smart in the sense of classical AI—that is they do not perform high level planning and problem solving. Instead they are smart in the sense that simple animals like insects are smart: able to robustly survive in and interact appropriately with their environments.

[1] The time delay is between 6 and 41 minutes.

1.4.3 Neural network control

A lot of research has been done on using feed-forward neural networks[2] as the adaptive component in a learning controller [86, 128]. The network weights can be adjusted using the backpropagation algorithm (a gradient descent approach, see Appendix C), genetic algorithms [8], or various stochastic search algorithms (for example, Alopex [124] and statistical gradient following [129]). Supervised training is usually performed using error signals derived from the system's performance error, although other approaches which transfer expert information from a rule base are common. For example, Handleman [43] trains a CMAC from a "knowledge base" to control a planar two-link manipulator. A similar approach is implemented in [107], and [52] uses a fuzzy rule base.

Several control approaches have been developed which perform training on the system with its controlling neural network unfolded over discrete time (see Appendix B). Backpropagation through time [95, 99] propagates error information backwards through time, and forward propagation algorithms [130] do the reverse. Such algorithms can also train recurrent neural network controllers that have their own dynamical properties. These algorithms have been generalized to continuous systems [96, 50] and made more efficient in various ways [24]. See Appendix B and [97] for more information.

Miller [82] has extended the backpropagation through time approach so that error information is also propagated through a custom-built central pattern generator (CPG). Judicious choice of the CPG circuit can improve the performance and stability of learning simple motor tasks.

Although theoretically elegant, forward and backward propagation approaches are ill suited to practical on-line control (see Appendix B). Others have used a more successful analytical control-theory approach to train a neural network so that it becomes an inverse (in some sense) of the system being controlled [102, 108, 89, 7].

1.4.4 Reinforcement learning (RL)

Reinforcement learning [55] (RL) is a description used in the literature for several different learning paradigms. In fact it more properly refers to a general class of problems, those in which the success or failure of a system is indicated by a scalar reward (or penalty) signal. The system must try to select its actions such that the total future reward is maximized. In the most difficult form of this problem the reward signal is a simple binary yes/no signal that is provided only after many actions have been performed.

One class of solutions to these problems use generalized dynamic programming approaches[3]. Examples are Sutton's temporal difference learning [117] and Watkins' Q-Learning [69]. These techniques can be very slow. This is because when complex behavior is required it is hard to formulate a reward measure which provides enough information about how the problem should be solved. For instance if the task is to learn to make a biped robot walk, a simple reward signal based on the distance covered before falling down will not indicate that walking is best achieved by taking one step at a time. Many RL techniques will be forced to try different actions at random until a successful sequence is generated. This becomes problematic as more complex behavior is required.

Despite this, many experimental robots use (quasi-) RL to adapt their behavior, such as [81] which

[2] Appendix C describes the most fundamental feed-forward neural network, the multi layer perceptron. [71] contains a lot of basic information about neural network architectures and learning algorithms. [49] contains revised and expanded information, including a complete derivation of backpropagation, temporal difference learning, generalization theory, radial basis function nets, dynamic and recurrent neural nets (with training algorithms). It has pseudo code for the more important learning algorithms. Also see Amari [5].

[3] For a description of dynamic programming, see Appendix D.

uses a modular feed-forward neural network to coordinate the basic reflexes of a robot so that it can adapt to an unknown indoor environment.

1.4.5 Eligibility-based RL techniques, and others

Another class of RL solution uses the "eligibility principle", which states that the controller parameters that have contributed to the current system state are the ones that need to change if the system state is incorrect. An eligibility value is associated with each parameter (the parameters are usually neural network weights), which indicates how much influence it has had on the control output, and thus its influence on the present system state. The concept of eligibility is one way to solve the classic AI "credit assignment" problem, i.e. to decide which parameters of a system get credit for responsibility for the system's actions. Eligibility values record the potential for some action-producing element to be changed by the reward signal (so that it is more likely to generate its action in the future). The eligibility model is based to some degree on biological models of classical conditioning (see section 2.3.2 and [6] for more information on classical conditioning). The FOX algorithm is a member of this class of RL techniques.

A classic example is Barto's adaptive heuristic critic [10]. In this technique an associative search element (ASE) makes associations between input sensors and the output action. An adaptive critic element (ACE) takes the binary penalty signal and makes it more useful by trying to anticipate the sensor states that lead to failure. Eligibility values are used in both elements to allow the penalty signal to affect a wider range of action-producing states. This allows states that lead to system failure to be discredited, making the corresponding incorrect control actions less likely in the future.

Other eligibility based RL methods include the drive reinforcement model [61], the constant prediction model [73], the Sutton/Barto [118] and time-derivative [119] methods, and SOP [126]. These models have varying degrees of biological realism and practical utility [72].

There are various methods which claim to be reinforcement learning because they follow Barto's ASE/ACE architecture, but which do not have an eligibility component. For example the stochastic real valued (SRV) reinforcement learning approach of [39] has an actor/critic architecture which uses neural networks with stochastic output nodes. It was trained using backpropagation on a robot peg-in-hole insertion task. Similarly Venugopal [125] uses stochastic search to train recurrent neural networks to learn the control dynamics of autonomous underwater vehicles.

Other methods claim to be reinforcement learning simply because some kind of training is performed. For example, [62] implements "reinforcement learning optimization" without any of the elements mentioned above, to learn the correct modulation of robot motor programs. [80] implements "complimentary reinforcement backpropagation" to train a neural network to control a modified toy car.

1.5 Summary

Having multiple interacting behavior-implementing modules is a useful design principle for "intelligent" robot controllers. The full potential of this paradigm can only be realized if the controller is able to learn about its environment. Many different learning methods have been explored, but this thesis will consider only one: eligibility based reinforcement learning, implemented in the FOX controller. It will be shown that FOX is an adaptive optimal controller in its own right, as well as a useful *learning component* for intelligent control systems.

Chapter 2

Biological motor control

2.1 Introduction

One of the great challenges of science today is to understand the human brain, and the biological basis of perceiving, learning, action and memory. The problem is, of course, the brain's incredible complexity of structure and subtlety of design. A great deal is known about the lowest levels: the biochemical mechanisms underlying the operation of individual neuron cells. Similarly the operation and interactions of various simple neural circuits in the brain is well understood. But at higher levels our knowledge becomes less detailed. Little is known about how the brain works at a "systems level", in other words how its various parts interact to produce coordinated, adaptive and intelligent behavior[1].

This chapter explains how the control of movement (motor control) is achieved in the brains of humans and other mammals. From an engineering perspective, the motivation for this study is to try and steal some of the good design features of the brain to use in a robot or "artificial life form". Only those parts of the brain concerned with motor control will be described here. Of course there are many other brain systems necessary for an organism's survival, such as sensory, memory and cognitive systems, which would also need to be emulated by a successful automaton. However these other systems will be largely ignored.

The overall structure of the brain is first outlined. Then the motor control system is described from the bottom up: the neurons, muscles, spinal cord, brain stem, cerebellum and motor cortex. This material is relevant in several respects.

- The CMAC and FOX are simple models of the cerebellum (FOX is more realistic in some ways).
- The simple model that will be constructed of the muscles and spinal reflexes will be used in Chapter 7 to achieve low level robot joint control.
- The intelligent controller design principle of multiple interacting behavior-implementing modules gets its biological justification here: the brain is structured in such a manner. The concept of brain-stem modulation of spinal reflexes is a good organizing principle for autonomous robot design and will be used in Chapter 7.
- The eligibility-based reinforcement learning technique gets its biological justification in the description of classical conditioning presented below: the Purkinje cells of the cerebellum implement

[1] The brains of lower (and therefore simpler) animals are better understood.

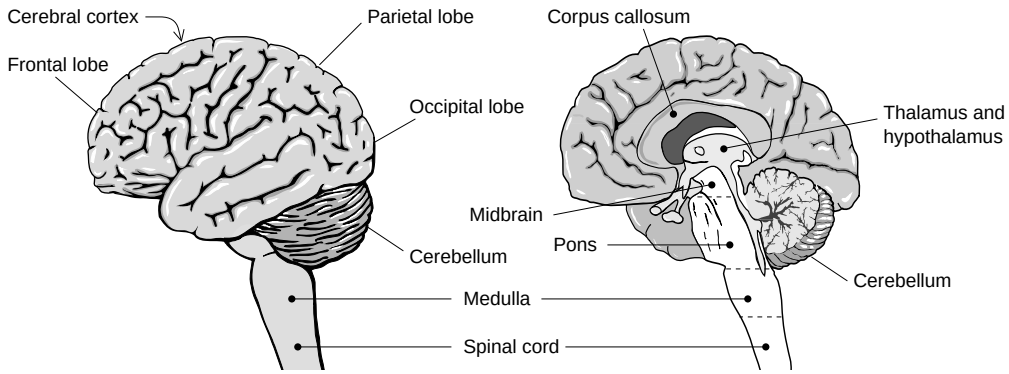


Figure 2.1: *The human brain, exterior (left) and interior (right) views.*

classical conditioning using a chemical eligibility signal.

There are generally two types of motor control theory: those that are hopelessly naïve from an engineering point of view, and those that are hopelessly naïve from a neurobiological point of view. The biological realism of the CMAC and FOX is not necessary from an engineering perspective, but it is interesting from a biology perspective.

Such a short chapter can not do justice to this huge subject area, so many details are omitted. Despite this, a lot of biological detail is presented, and the uninterested reader may skip to the conclusions at the end of this chapter. Much of the information presented here comes from [59], [53], [1] and [40][2]. At each level a simplified functional model will be presented which encapsulates the essential features (and fills in some details where there is no supporting biological evidence).

2.2 Overall structure

The brain contains many subsystems specialized to different tasks, much like organs in the body. Human behavior (or that of any animal) arises from the interaction of these systems. The gross anatomy of a human brain is shown in figure 2.1. Figure 2.2 shows the principle structures involved in motor control and the connections between them. The motor systems are arranged in a rough hierarchy. Upper levels of the hierarchy send modulatory commands to the lower levels, and the lower levels in turn send back processed sensory and state information.

The *spinal cord* is the major interface between the brain and the outside world. It contains many bundles of sensory and motor nerves, and also many neural circuits organized to provide reflex behaviors.

The *brain stem* is made up of the medulla, pons, and mid-brain. It contains many distinct nuclei (groups of neurons) which are specialized to various tasks such as maintenance of posture, control of balance and transformation of sensory information. It regulates many of the low level aspects of behavior, and implements basic survival motor programs.

The *cerebellum* helps to coordinate instinctive and learned motor behaviors. It receives information from the motor cortex and from the spinal cord, which it uses to modulate other brain stem areas.

The *thalamus* processes and distributes almost all sensory and motor information going to the cerebral cortex. The *basal ganglia* are an adjunct to the cerebral cortex, they have a role in the coordination of

[2] Also see [101] and [9] for general information on brain structure and dynamics.

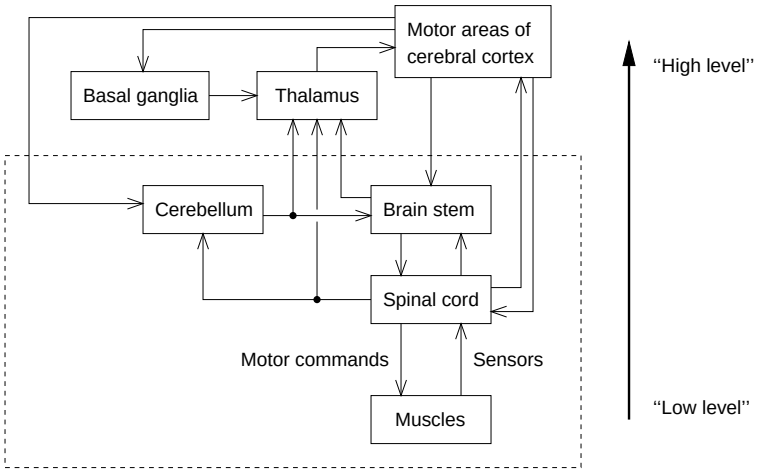


Figure 2.2: *The principle motor control areas of the brain.*

movement and also play a part in cognition. They may be involved in computing the transitions between different movement states [28].

The *cerebral cortex* is a thin folded sheet of neurons which covers the cerebrum. The cerebrum has two hemispheres on either side of the head which are connected together by a thick bundle of nerve fibers called the corpus callosum. The cerebral cortex is concerned with perceptual, cognitive, and higher motor functions, as well as emotion and memory. Complex behavioral responses originate here. It has many areas which are specialized to different cognitive functions, such as vision, language, and planning.

Figure 2.3 shows the motor control areas of the brain in more detail. Not all the nuclei or connections between them are shown, just the most well known. This figure highlights the fact that the brain is very elaborate and has many interacting subsystems.

2.3 Neuron biology

Neuron biology is a complicated subject, and only the barest details will be given here. A neuron is a cell that is specialized to carry electrochemical signals. The signals are represented as voltage differences across the cell membrane. Figure 2.4 shows the basic structure of a neuron. The neuron has many thin projections that connect it to other neurons. These are the *dendrites* which carry inputs to the neuron, and the *axons* which carry the neuron's output to other neurons. Over short distances the static neuron voltage can carry information. Over longer distances signals are transferred along dendrites or axons as a stream of voltage pulses, and information is encoded in the pulse rate (also called the firing rate).

Each pulse transmitted to a neuron is integrated by the cell body until a threshold is reached, whereupon the neuron emits its own pulse and the integration restarts. The interface between any two neurons occurs at a synapse, which is a junction between an axon terminal and a dendrite. The axon terminal can be excitatory or inhibitory by some amount, which means that the synapse modulates the incoming pulse so that it will have a greater or lesser integrating effect on the target neuron, thus increasing or decreasing its firing rate.

Neurons come in many shapes and sizes. For instance, the granule cells of the cerebellar cortex are tiny, star shaped and only have four inputs (synapses) on average. In contrast the Purkinje cells are much larger, tree shaped, and have up to 200,000 inputs each. Neurons vary greatly in their internal dynamics:

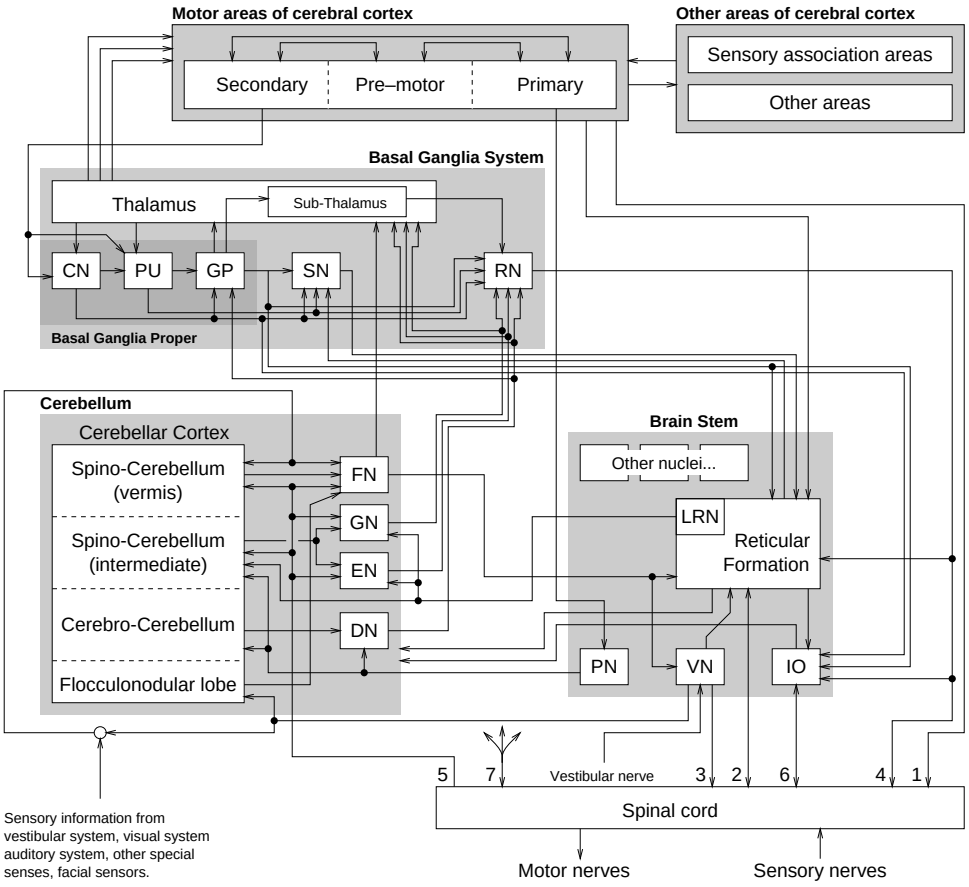


Figure 2.3: *The major motor control areas of the brain, in more detail. FN: Fastigial nucleus, GN: Globose nucleus, EN: Emboliform nucleus, DN: Dentate nucleus, RN: Red nucleus, PN: Pontine nuclei, LN: Lateral reticular nucleus, CN: Caudate nucleus, PU: Putamen, GP: Globus Pallidus, SN: Substantia nigra. Spinal cord tracts: 1:cortico-spinal tract, 2:reticulo-spinal, spino-reticular, and various other tracts, 3:vestibulo-spinal tracts, 4:rubro-spinal tract, 5:spino-cerebellar tracts, 6:olivo-spinal and spino-olivary tracts, 7:various other tracts. This picture was assembled from several sources, including [59], [53], [1] and [40].*

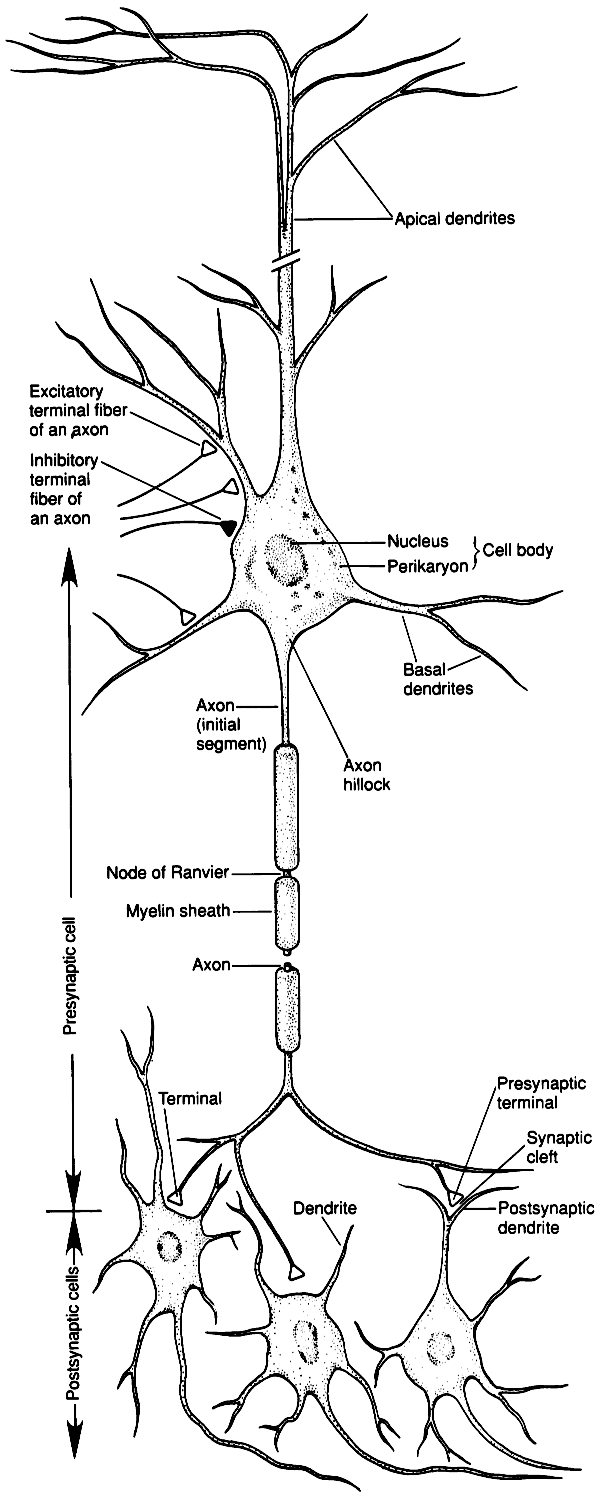


Figure 2.4: *The basic features of a typical neuron. From Principles of Neural Science, 3rd edition, by E.R. Kandel, J.H. Schwartz and T.M. Jessel, page 19. Copyright ©1991 Appleton & Lange.*

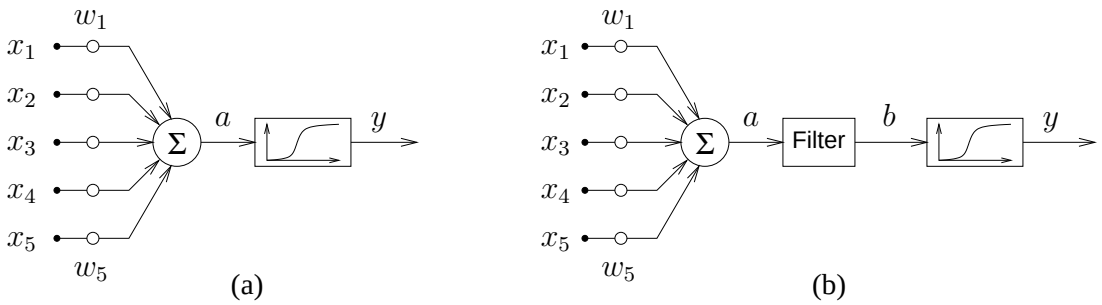


Figure 2.5: *Some simple neuron models.*

some just act to relay incoming signals, while others can have complicated oscillatory behavior that is modulated by the incoming signals.

2.3.1 Neuron models

Despite this complexity it is common to use extremely simple neuron models in artificial neural networks. Figure 2.5a shows a simple stateless neuron model. The input and output values represent firing rates. The weighted inputs are summed and then a "squashing function" $f(\cdot)$ is applied to get the output:

$$y = f\left(\sum_i w_i\, x_i\right) \tag{2.1}$$

Note that the weights w_i represent the synapses, with $w_i > 0$ being excitatory and $w_i < 0$ being inhibitory. The squashing function is usually bounded above and below and monotonically increasing. A typical example is the sigmoid function:

$$f(a) = \frac{k_1}{1 + e^{-k_2(a-k_3)}} \tag{2.2}$$

where $k_1 \ldots k_3$ are arbitrary constants. Figure 2.5b is similar except that the output goes through some kind of differential equation, typically a first order filter:

$$\frac{\mathrm{d}\, b}{\mathrm{d}\, t} = \tau(a - b) \tag{2.3}$$

where τ is a time constant. This adds state variables to the system and allows the modeling of more dynamic neural networks. More sophisticated models of neurons and neural groups, and a description of their information processing capabilities are given in [98] and [45].

2.3.2 Models of learning

Learning can be regarded as the neural process of forming internal *representations* of the external world. Different parts of the brain perform different types of learning, using various mechanisms. In most cases learning happens at the level of individual neurons by adjustment of synapses or alteration of a neuron's dynamics in response to incoming signals.

Classical conditioning is a common mode of associative learning, in which stimuli are associated with an appropriate response. Here is a simple (contrived) example: if a person stubs their toes against

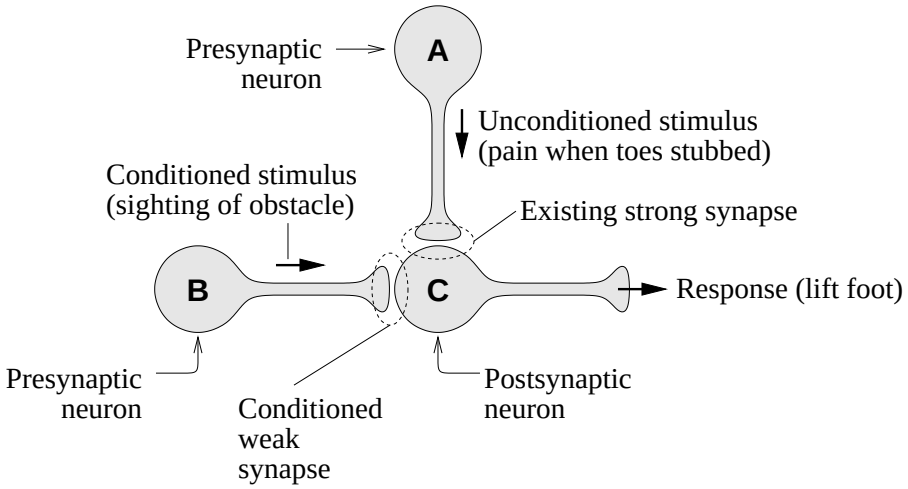


Figure 2.6: *The mechanism of classical conditioning.*

an obstacle while walking, they will withdraw the foot from the obstacle. If the person is able to see the obstacle as well, classical conditioning can associate that sighting with the pain resulting from a collision, and *preemptively* trigger the withdrawal of the foot to avoid the obstacle.

In 1949 Donald Hebb proposed a cellular mechanism for classical conditioning, which is now known as Hebbian learning [44] (figure 2.6). An "unconditioned stimulus" (the pain signal when the toes are stubbed) originates from neuron A. This is able to elicit a strong response in neuron C (which lifts the foot above the obstruction) because of the high-strength synapse coupling the two. A "conditioned stimulus" (the visual sighting of the obstruction) originates from neuron B. It is initially unable to elicit a response because its synapse on to C is weak. However, if B fires *at the same time* as C is firing, then the B-C synapse will be strengthened. Initially C only fires in response to A. If B fires in conjunction with A then B's synapse will grow stronger and it will also be able to trigger a response in C. In this way the conditioned stimulus is *associated* with the unconditioned stimulus.

The synaptic mechanisms of Hebbian learning have been extensively studied [58]. The way in which the conditioned and unconditioned stimulus are paired in time is significant. They have to be close enough together so that C's firing is "remembered" by the synapse when B fires. Note that classical Hebbian learning only results in synaptic strengthening. Other (similar) processes allow the synapse to be weakened as well.

Other neuron learning modalities include long and short term potentiation, sensitization and habituation [59]. These all use mechanisms similar to Hebbian learning. These synaptic learning rules are known to occur in the neurons of the cerebral cortex, the cerebellar cortex, and the hippocampus.

2.4 Muscles

The actuators of the motor control system are the skeletal muscles. Every joint in the body has two or more muscles which can apply a torque to the joint by contracting (figure 2.7). A simple muscle model is a spring whose spring constant can be changed by external signals. Muscles are organized in extensor/flexor pairs, so that movement in both directions around a joint can be achieved. Different muscles that work to provide the same mechanical action on a joint are called synergists. Muscles that

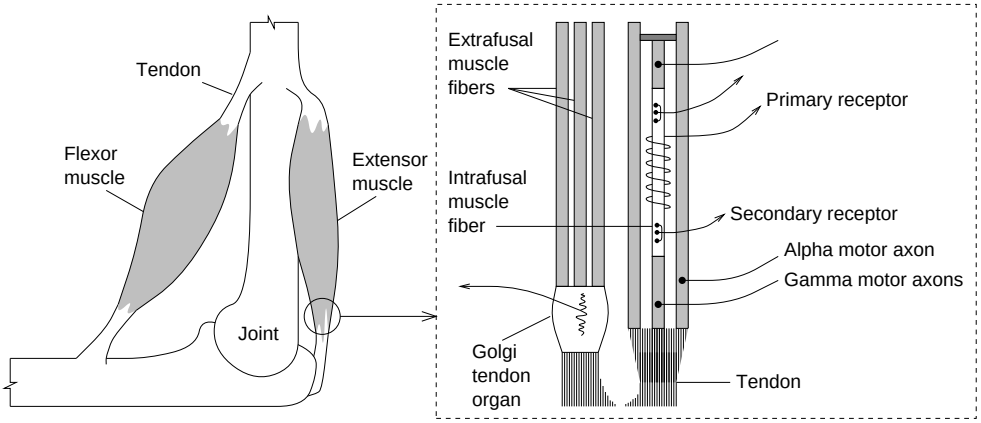


Figure 2.7: *Muscle anatomy and a schematic of the internal fiber arrangement.*

work to provide opposing motions are called antagonists. Different muscles differ in size, speed, and fatigue properties. Muscles are made up of many small cylindrical fibers, of two main types. The large extrafusal fibers provide the main contracting force. The smaller intrafusal fibers act as sensory organs.

2.4.1 Muscle contraction

Extrafusal fibers are innervated by the axons of large alpha motor neurons. The fiber contracts briefly when it receives a pulse from the axon. The alpha motor neurons are contained within the spinal cord. Each one drives from about 10 to 1000 muscle fibers, collectively referred to as a motor unit. The contraction force of a muscle is controlled by modulating the firing rate of the motor axons and by selectively activating more motor units as the desired force increases. As the higher levels of the brain request more muscle force, more motor units are recruited in order from the weakest (but most energy efficient) to the strongest (but least energy efficient). This allows a fine grading of the muscle force at the least metabolic cost.

2.4.2 Muscle spindles

The intrafusal fibers are attached to small gamma motor neurons (figure 2.7). They provide little muscle power—instead they function as sensory organs. The central area of the fiber contains two kinds of sensors. Primary receptors measure the length plus rate of change of length of the central area. Secondary receptors measure the length only. Both receptors send nerve fibers to the spinal cord. These receptors are stimulated whenever the central area is stretched. This happens either when the muscle itself is stretched (because the intrafusal fibers are tied to the extrafusal fibers), or when the intrafusal fiber tries to contract by itself against its attachments. In this way the intrafusal fibers act as comparators. The signals received from the stretch receptors can thus be modulated by the firing rate of the gamma motor neurons.

The stretch dynamics of the intrafusal fibers are complex, nonlinear and subject to both plastic and elastic effects. The dynamics of the spindle receptors are similarly complex, as shown in figure 2.8. The primary receptor responds quickly (under 1ms) to the initiation of any stretch, with a high firing rate. Thereafter its signal is roughly proportional to the length of the central area (plus the derivative of this length). The secondary receptor responds more slowly.

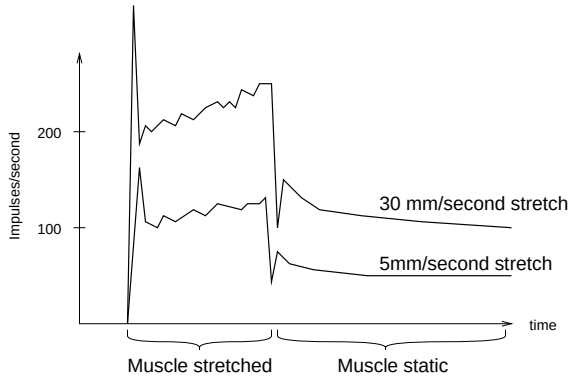


Figure 2.8: *Approximate primary receptor response for different muscle stretch rates (From [59] page 569).*

There are two types of gamma motor neuron, static and dynamic, which connect to intrafusal fibers with different dynamic properties. Activation of the static gamma system increases the steady state primary receptor response and the secondary receptor response. Activation of the dynamic gamma system increases the transient primary receptor response. The static and dynamic gamma activity (that is, the gain of the static and dynamic elements of the primary receptor response) is varied by higher level brain centers according to the current behavior. Both types of activity increase with the speed and difficulty of the movement. Static (but not dynamic) gamma activity is required for steady postures such as sitting or standing. Dynamic gamma activity is required for faster movements, such as walking, running, or difficult balancing tasks.

When the muscle contracts, the intrafusal fibers don't just go slack. Instead the gamma system is activated concurrently to shorten them and maintain tension. This alpha-gamma co-activation is controlled by the spinal cord.

2.4.3 Golgi tendon organs

Golgi tendon organs occur at the interface between the tendon and the body of the muscle (figure 2.7). Each of these organs is in series with about 10–15 muscle fibers. They measure tension in the muscle. Their receptor response is similar to that of the primary receptors in the muscle spindles, except that the quantity measured is muscle tension, not intrafusal fiber stretch.

2.5 Spinal cord

The anatomy of the spine and spinal cord is shown in figure 2.9. Almost all of the body's peripheral sensory and motor nerves connect to the spinal cord at some point. Sensory nerves (from various sense organs in the muscles, skin, and other places) enter the spinal cord through the sensory roots. Motor nerves to the muscles originate from spinal cord neurons and leave through the motor roots.

The white matter of the spinal cord contains many tracts of nerve fibers. These tracts convey ascending and descending fibers to other spine segments, and between spine segments and the upper brain. For example, the corticospinal tract contains fibers which connect motor neurons directly to the cerebral cortex. But the spinal cord is not just a passive cable. Its gray matter contains neural circuits which

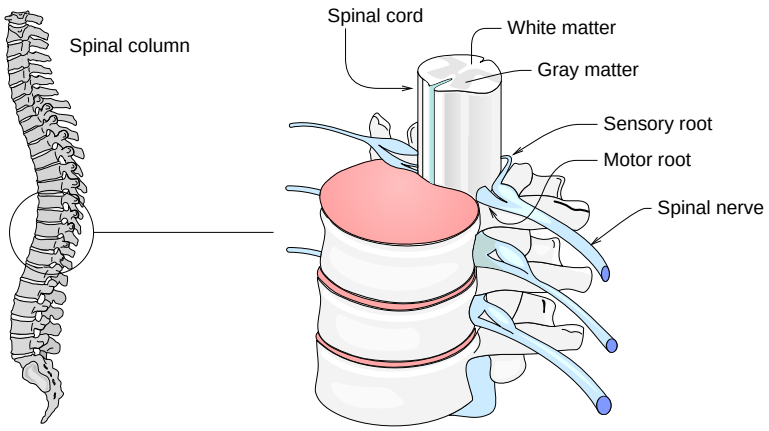


Figure 2.9: *Anatomy of the spine and spinal cord.*

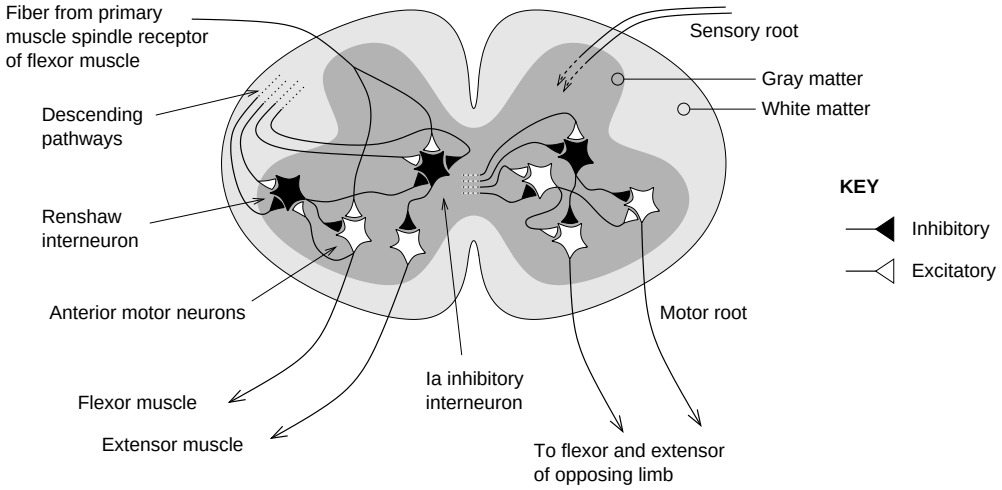


Figure 2.10: *Some basic spinal cord circuits.*

implement many of the reflexes that are required for the low level control of movement. “Reflexes” in this context are typically fast stereotyped responses that are triggered by environmental stimuli.

There are two main classes of spinal cord neurons: large anterior motor neurons and small internuncial cells (figure 2.10). All motor root nerves originate from the anterior motor neurons. The internuncial cells are much more numerous than the motor neurons. They are interconnected in complex patterns and they implement many simple (and not-so-simple) reflexes. Most incoming sensory signals connect to the internuncial cells and are able to trigger various reflexes. Figure 2.10 shows a few of the basic spinal cord neural circuits.

Most of the spinal cord reflex circuits are modulated by higher brain centers via the descending tracts. This modulation is very important, as coordinated movement can only be achieved by selectively sensitizing appropriate reflexes and suppressing others, depending on the situation.

2.5.1 Stretch reflex

The simplest (and arguably most important) spinal reflex is the stretch reflex. This reflex opposes the stretching of a muscle beyond its present length. It is implemented by having the primary muscle spindle receptors send excitatory signals directly to the anterior motor neurons for synergist muscles (figure 2.10). When a muscle stretches, the primary spindle receptors respond immediately, exciting the motor neurons and causing the muscle to contract. Because the stretch reflex operates in all muscles, any movement of the joint will be opposed.

The negative stretch reflex opposes shortening of the muscle in a similar manner. All muscles have a small resting tension, so the primary receptors are always active when the muscles are holding a static position. When the muscle is shortened, the primary receptor response drops, and this causes excitation of the antagonist muscles.

The effectiveness of the stretch reflex can be modulated by signals from the descending pathways. This is obviously necessary, because if the stretch reflex was always fully active the body would be frozen in a static posture. This modulation may occur through "pre-synaptic modulation", where a synapse from a descending axon directly inhibits the synapse from the spindle primary receptor to the anterior motor neuron. This modulation allows higher brain centers to achieve fine control of movement. The stretch reflex gain can also be altered by changing the activity of the gamma motor neurons.

The gamma fibers in conjunction with the stretch reflex function as a kind of servo system. When gamma neurons are excited the ends of the intrafusal fibers contract, causing the primary receptors to be excited. This excites the stretch reflex and causes contraction of the muscle until the central area of the intrafusal fibers are no longer stretched. The muscles contract to a length predetermined by the gamma fiber activation. The dynamics of this contraction can be altered by changing the amount of static gamma versus dynamic gamma activation. This servo system is controlled by higher brain centers, principally the cerebellum, motor cortex and the basal ganglia, via the brain stem nuclei.

Alternatively, when the gamma system is inactive, the intrafusal fibers are flaccid and the alpha system is the main source of muscle contraction. The alpha mechanism is a direct muscle drive with no limits on the extent of movement. However, during fast muscle contractions both the alpha and gamma neurons can be excited together so that muscle spindle feedback does not disrupt the movement.

Muscle tone is the static tension in a muscle, and is present even when the muscle is at rest. Muscle tone is caused partially by the intrinsic elasticity of the muscle and partially by the stretch reflex.

2.5.2 Reciprocal inhibition reflex

Whenever synergist muscles contract, the antagonist muscles must relax for the joint to move. To accomplish this, all neural circuits which excite a motor neuron also send signals to internuncial type-Ia inhibitory interneurons. These interneurons send inhibitory signals to the motor neurons for antagonist muscles. Thus opposing muscles groups will cooperate to achieve movement. The reciprocal inhibition circuitry for the stretch reflex is shown in figure 2.10, where the primary muscle spindle receptors also send branches to the inhibitory interneurons.

Commands from higher centers can regulate the joint stiffness by controlling the degree to which the inhibitory interneurons are effective. For example, inhibition of an inhibitory interneuron will result in less motor neuron inhibition and thus more joint stiffness as antagonist muscles work against each other.

2.5.3 Tendon reflexes

Axons from the Golgi tendon organs synapse directly on to the anterior motor neurons for synergist muscles, implementing something similar to the stretch reflex.

Golgi organ excitation can also cause *inhibition* of the corresponding motor neurons, via inhibitory internuncial cells. This is organized so that extreme muscle tension causes inhibition and therefore muscle relaxation. This is a protective reflex to prevent tearing of the muscles and tendons during exertion.

The tendon reflex is a feedback circuit that can maintain muscle tension at a certain value. The set-point for this reflex is controlled by higher brain centers. This is useful for tasks that need a constant muscle force, not a constant muscle length. This reflex is required because a muscle's force dynamics are quite nonlinear (the force exerted for a given excitation depends nonlinearly on the muscle's length).

2.5.4 More complex reflexes, and central pattern generators

More complicated reflexes involve multiple interneurons and coordinated groups of muscles. In these reflex circuits, some interneurons act as "gates" which allow higher centers to modulate reflexes. These either use pre-synaptic modulation (effectively a synapse on a synapse) or direct inhibition of neurons in reflex circuits.

Central Pattern Generators (CPGs) are oscillatory neural circuits which control rhythmic movements such as scratching and walking. Many CPGs work because of reciprocal inhibition and reflex rebound. Reciprocal inhibition means that groups of neural circuits in the CPG associated with different muscles inhibit each other. Reflex rebound means that after a reflex has been triggered it can become harder to trigger for a given time thereafter, but the corresponding opposing reflex becomes easier to trigger.

Many CPGs rely on the complicated internal dynamics of individual neurons. For example, a neuron can increase its firing threshold with extended excitation (this is called adaptation). Similarly, a neuron's threshold can increase as a result of past inhibition (this is called rebound). Some CPGs use pacemaker cells which have an oscillating output, i.e. they repeatedly generate short bursts of pulses.

2.5.5 Postural and locomotion reflexes

Complex groups of internuncial cells are organized to provide reflexes that are useful for maintaining postural equilibrium, and in walking. Multiple spinal cord segments communicate via the spinal cord tracts, to coordinate multi-segment reflexes which involve large parts of the body. These reflexes have been experimentally isolated in quadrupeds, particularly cats and dogs [40].

Many locomotion reflexes are controlled by CPGs. The CPGs cause alternating movement of muscles on opposite sides of the body. There are independent CPGs for each limb, which are coupled together to achieve coordinated whole-body behavior.

The *posture supportive* reaction causes a limb to extend against pressure applied to a footpad. This reaction allows a quadruped to stand statically.

The *magnet* reaction causes a foot to move in the direction from which pressure is applied to the edge of a footpad. This is useful for keeping balance.

Stepping reflexes cause oscillations of the limbs in appropriate walking patterns. For example, the reciprocal stepping response in quadrupeds causes adjacent legs to go forwards and backwards rhythmically. The diagonal stepping response causes coordinated stepping movements in all four legs. Galloping movements can be triggered by simultaneous stimulus on the front or back paws, as would be felt during normal galloping. These reflexes result mainly from reciprocal inhibition and rebound effects.

Spinal *righting reflexes* in decerebrate cats and dogs cause them to make movements appropriate to standing up when they are placed on their side.

The *scratch reflex* is particularly sophisticated. In quadrupeds it is triggered by skin irritation and it causes a leg to find and rhythmically scratch the irritated site.

2.5.6 Other spinal reflexes

Transient changes in motor neuron output are opposed by recurrent inhibition with *Renshaw interneurons*. Renshaw interneurons are excited by a motor neuron. They inhibit that same motor neuron and the inhibitory interneurons for the antagonist muscles (figure 2.10). This effectively limits the speed at which the motor neuron output can change. Thus Renshaw cells act as a low pass filter for all motor commands [110]. The effect is distributed to muscles around a joint by type-Ia interneurons. Renshaw cells are modulated by higher centers to change the effective "excitability" of a muscle's motor units.

Group Ib inhibitory interneurons have many excitatory and inhibitory inputs from other internuncial neurons. They inhibit the anterior motor neurons. These neurons mediate the tendon reflex, the withdrawal reflex, and many others.

The *withdrawal reflex* ("flexor" reflex) makes the limb withdraw from a painful skin stimulus. A complex circuit of internuncial cells is involved, and the stimulus is distributed to many muscles. The response is maintained for some seconds after the stimulus.

The *crossed extensor reflex* occurs 0.2–0.5 seconds after a stimulus triggers the withdrawal response. This causes the opposing limb to extend, pushing the body away from the stimulus.

Autonomic reflexes control other body functions like sweating, vascular tone, motor functions of the gut, and so on. These will not be considered further.

2.5.7 Spinal motor model

What are the variables of low level muscle control? In other words, what are the "joint commands" that descend from the high level systems to instruct the joints? At the very lowest level it could be said that the variables of muscle control are simply the alpha motor neuron activations which produce joint torque directly. But more convenient variables can be found.

Latash [65] has suggested that the set-point of the stretch reflex is an appropriate control variable. This conclusion is drawn from Merton's servo hypothesis, which is that the main function of the stretch reflex is compensation of the influence of load upon muscle length, i.e. that it is a length regulating mechanism. This leads to the low level joint controller model shown in figure 2.11. This model is a simple proportional-derivative controller that tries to get the joint angle θ equal to a reference θ_r by applying an appropriate joint torque. The constants k_p and k_d set the "spring constant" and amount of damping in the controller. They are equivalent to the static and dynamic gamma motor neuron activations. A torque output filter performs a role equivalent to the Renshaw cells by limiting the rate of change of torque. The parameter τ adjusts the filter's time constant.

Specifying θ instead of the torque simplifies the control of movement, because even with static parameters it can provide joint movements with reasonably good dynamics. It allows complex movements to be "played out" in θ coordinates by higher level centers, effectively giving those centers less to think about.

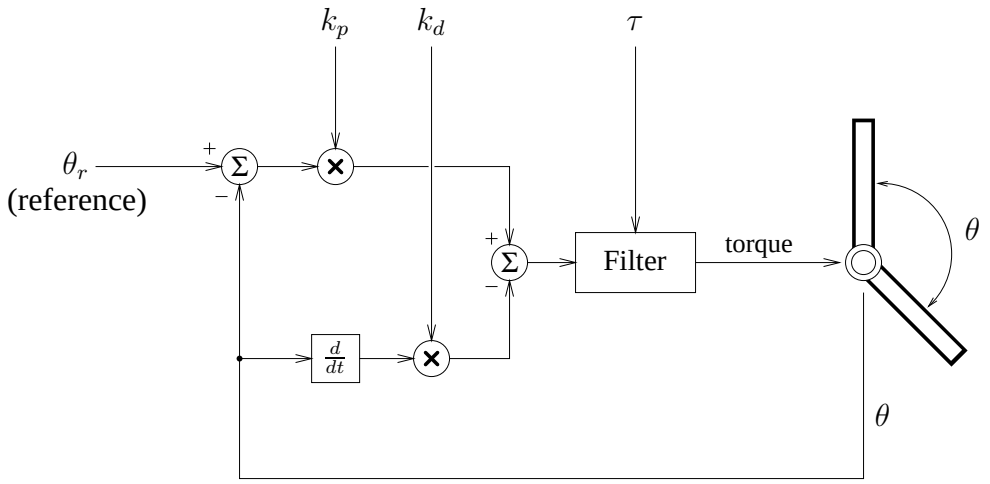


Figure 2.11: *Low level joint controller.*

2.6 Brain Stem

The brain stem is a complex extension of the spinal cord. The basic principle of multiple interconnected modulated reflexes is also implemented in the brain stem, but highly elaborated to handle more complex situations, and to use past experience to improve performance.

The brain stem contains many nuclei (groups of neurons) with specialized functions. It also contains the more diffuse "reticular formation", which controls many discrete motor activities, such as the control of equilibrium, the support of the body against gravity, and high level reactions to prevent the body from falling.

Many stereotyped movements are controlled by the brain stem. In humans these include standing still, sitting up, turning, head tracking movements, chewing, swallowing, yawning, sucking the fingers, and so on. All of these behaviors are necessary for survival, which is why they are "hard wired" into the brain and not learnt.

2.6.1 Posture and equilibrium

Maintenance of postural equilibrium is dependent on muscle sensors, the vestibular system, and vision. The overall postural response depends on experience, because the correct reaction to maintain equilibrium is highly dependent on the body state and the environment.

The vestibular system helps to maintain the body's balance with a set of "righting reflexes". These reflexes move the limbs appropriately to compensate for falling off balance. The sensors of the vestibular system are the semi-circular canals and utricle of the inner ear. The three semi-circular canals detect angular accelerations of the head in three planes. The utricle detects linear accelerations of the head and the head's orientation with respect to gravity. The output of the vestibular sensors connects to the vestibular nuclei, various other brain stem regions, and the cerebellum. Neck muscle sensors are integrated with vestibular information to determine the whole-body configuration.

2.7 Cerebellum

The cerebellum's task is to regulate the motor activity occurring in other brain areas such as the spinal cord, brain stem, basal ganglia and motor cortex (figure 2.3). The cerebellum is essential for the coordination, dexterity and timing of almost all body movements, especially high speed movements. Signals originating from the cerebellum modulate the extent of movement, initiate and terminate movement, and precisely control the timing of the many events within coordinated sequences of movement. In conjunction with the vestibular system it also helps to ensure correct balance. It can convert "clumsy" movement commands originating in the motor cortex into smooth, fluid actions.

The cerebellum makes up only 10% of the brain volume, but it contains more than half of all the neurons [59, p. 627]. The exterior of the cerebellum is its cortex, a thin sheet of neurons that is folded in on itself many times to pack as much surface area as possible into the small space (about $500\ \text{cm}^2$). Surrounded by the cortex are three pairs of nuclei which serve to relay information from the cortex to the rest of the brain (figure 2.3).

The cerebellum is one of the principle adaptive components of the motor system. Its goal is to modulate motor commands so that *actual* movements match *desired* movements. To achieve this it is continuously being trained to anticipate disturbances and preemptively correct for them.

2.7.1 Cerebellar cortex: Anatomy

The cerebellum is one of the best studied regions in the brain [1, p. 194], so a great deal is known about its structure and operation. Figure 2.12 shows a slice of the cerebellar cortex. The cortex has three layers of neurons which contain five different cell types (the "white matter" layer in the figure contains only axons).

The mossy fibers originate in other brain stem and spinal cord nuclei. They are the major inputs, encoding all of the sensory information that the cerebellum needs. The mossy fibers terminate on granule cells. The granule cells are tiny and very numerous (there are about 10^{10}–10^{11} of them). Each one takes on average four mossy fiber inputs. Their axons are called "parallel fibers" because they run up in to the molecular layer, split into two and run parallel to each other up and down the cortex.

The Purkinje cells are much larger than the granule cells, and there are only about 15 million of them. They have extensive dendritic trees which collect inputs from up to 200,000 parallel fibers [59]. The Purkinje cell axons are the cortex output. They inhibit neurons in the cerebellar nuclei.

The climbing fibers originate from the "inferior olive", a nucleus in the brain stem. They carry training information to the Purkinje cell synapses by wrapping around the Purkinje cell dendritic tree. Training signals on the climbing fibers change the strength of the Purkinje cell synapses.

The Golgi, stellate and basket cells take parallel fiber inputs and serve to regulate the firing rates of the other cortex neurons by inhibition. The Golgi cells inhibit the granule cells, and the stellate and basket cells inhibit the Purkinje cells.

Different areas of the cerebellar cortex serve different functions. The vestibulo-cerebellum governs eye movement and body equilibrium. The spino-cerebellum controls detailed motor actions. It takes most of its input from the spinal cord. It contains somatosensory maps of the body, which means it is divided into regions that are specific to different body parts. The cerebro-cerebellum connects to the motor cortex via the thalamus and various nuclei. It is thought to aid the planning and initiation of movement.

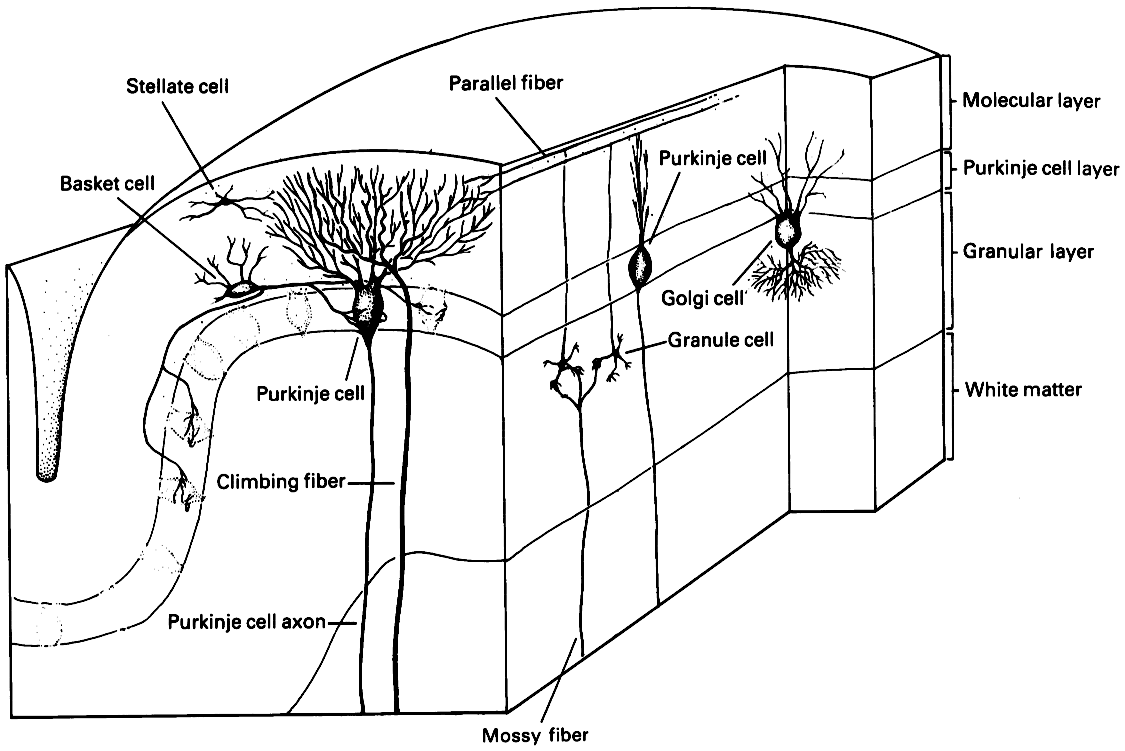


Figure 2.12: *The cortex of the cerebellum, which has three layers and contains five kinds of neurons. From Principles of Neural Science, 3rd edition, by E.R. Kandel, J.H. Schwartz and T.M. Jessel, page 630. Copyright ©1991 Appleton & Lange.*

2.7.2 Cerebellar cortex: Operation

Figure 2.13 shows a schematic of the neurons and connections within the cerebellar cortex and related nuclei. The inputs to the cerebellum are mossy fiber signals which contain sensory information. Some inputs are processed sensory information from high level visual, auditory and motor cortex areas. Other inputs come from low level spinal and brain stem areas. The cerebellum's own motor commands also re-enter the cortex via the lateral reticular nucleus. The area of the cortex that controls a muscle group has inputs from the motor cortex area controlling the same group, and also from the corresponding spinal sensors.

The Purkinje cell outputs modulate the spinal cord reflexes and motor neurons through the cerebellar nuclear cells. Thus the cerebellum is able to control the extent, speed, stiffness and timing of movements.

One hypothesis of cerebellum operation is that it functions as an array of adjustable pattern generators [86, p. 301]. Figure 2.13 shows that cerebellar nuclei (CN) cells, red nucleus (RN) cells and lateral reticular nucleus (LRN) cells are connected in rings with excitatory synapses. It was proposed that these rings function as bistable devices which can be switched on and off by the Purkinje cell inhibition. Multiple interacting bistable rings would form a "state machine" which is responsible for the execution of a motor program. This state machine would be modulated by the Purkinje cells but is otherwise self sustaining. This is an attractive idea, but it is probably not the whole story. Other studies [23] have shown the importance of the cerebellum in the *timing* of motor responses.

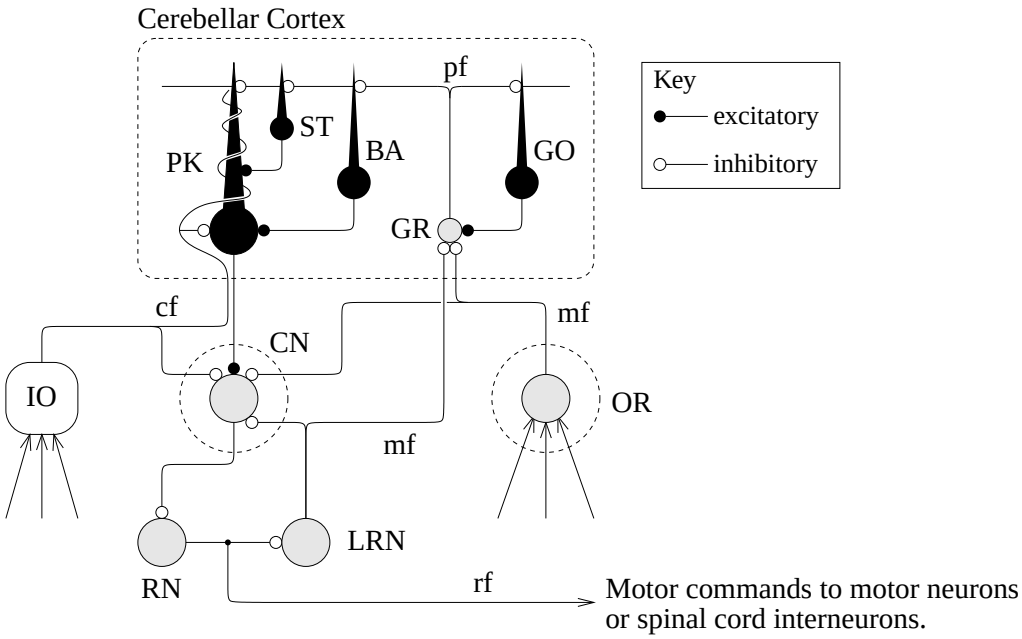


Figure 2.13: *Schematic of the neurons and connections within the cerebellar cortex and related nuclei. Key: Purkinje cell (PK), stellate cell (ST), basket cell (BA), granule cell (GR), Golgi cell (GO), originating cell (OR), cerebellar nucleus cell (CN), inferior olive (IO), red nucleus cell (RN), lateral reticular nucleus cell (LRN), mossy fiber (mf), parallel fiber (pf), climbing fiber (cf), rubrospinal fiber (rf).*

Figure 2.14 shows a simple mathematical model of the cerebellar cortex, which represents the two most important cell types: the granule cells and the Purkinje cells. The Purkinje cells just perform a weighted sum of their granule cell inputs.

In this model the granule cells are arranged so that the synapses of the Purkinje cells are used most effectively. Each granule cell forms a logical AND of its inputs (so it only fires when all of its inputs are above a certain threshold). This allows pre-wired associations to be formed among groups of sensor inputs. For example, in figure 2.14, if only one of A and B can fire at once (similarly for C and D) then the granule cells G1. . .G4 represent all possible input combinations. This gives the Purkinje cells maximum flexibility, because they can have a different response for every possible input (each response is determined by a different synapse).

A fundamental requirement is that the granule cell outputs can distinguish between different body configurations. In the best case the granule cells are configured so that any cerebellar input triggers only a small percentage of them, and as the input changes the active granule cell group also changes. This gives the Purkinje cells the maximum power to discriminate between different inputs. It also maximizes the effectiveness of training, because Purkinje cell learning in one input configuration will not degrade the information stored in another area if there is no overlap in the corresponding granule cell activations.

Remember that the Golgi cells inhibit groups of granule cells. It has been hypothesized that the role of the Golgi cells is to keep the firing rate of the granule cells constant [46]. It is further hypothesized here that the reason Golgi cell inhibition allows only a small number of granule cells to be active at any one time is to force them to compute the logical AND of their inputs (or something similar). This is supported by studies such as [23] which show that Golgi cell feedback acts to limit granule cell activation.

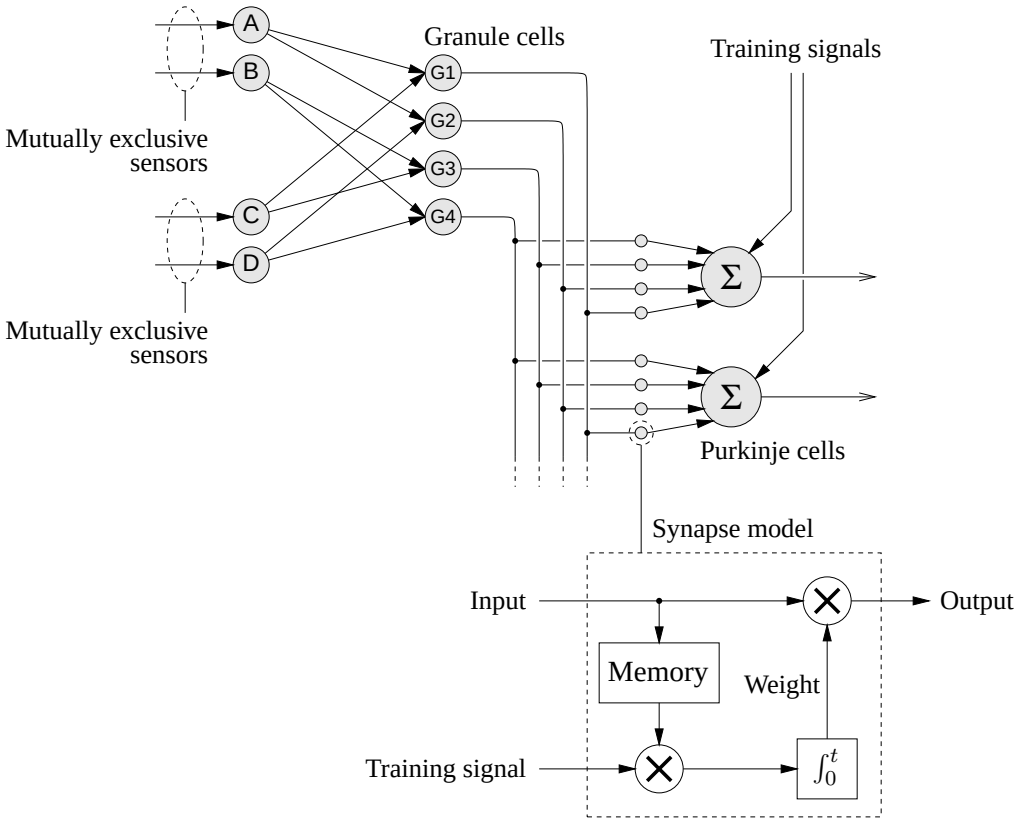


Figure 2.14: *A model of the cerebellar cortex, incorporating the granule and Purkinje cells. This model also implicitly incorporates the Golgi cells, as each granule cell computes the logical AND of its inputs. Each Purkinje cell computes a weighted sum of its inputs.*

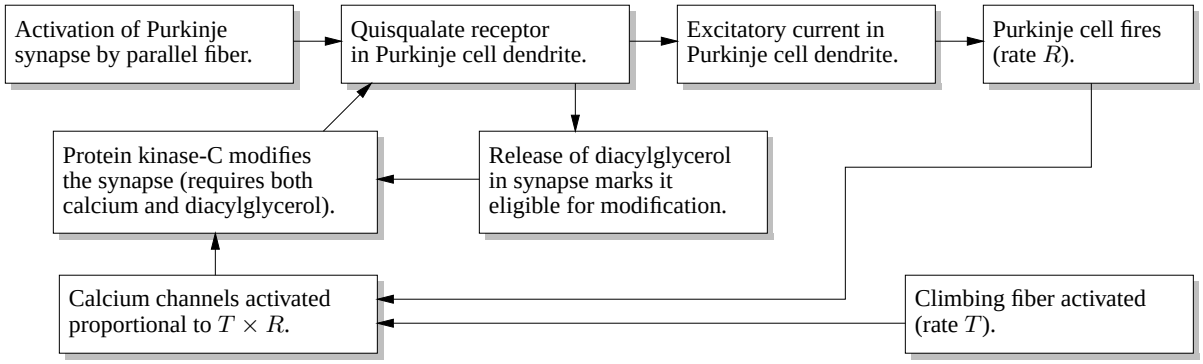


Figure 2.15: *The biochemical process by which Purkinje cell synapses are trained (from [86]).*

It is possible that the "command" inputs from the motor cortex are also ANDed with the other cerebellar inputs to allow the motor cortex to select different motor commands or motor programs.

2.7.3 Cerebellar cortex: Training

The purpose of cerebellar training is to correct mismatches between intended movements and the results achieved. The synapses of Purkinje cells undergo classical conditioning via a variant of the Hebbian learning mechanism. The parallel fibers transmit the conditioned stimulus, and the climbing fibers transmit the unconditioned stimulus. The climbing fibers have strong synapses on to the Purkinje cells, i.e. they are easily able to trigger a response. Climbing fiber signals convey information about unexpected events, or alternatively they have the meaning "do more (or less) of this in the future". For example, if a limb bumps into an unexpected obstacle during a motion, climbing fiber signals will be generated to move the limb away from the obstacle. Climbing fiber signals come (though the inferior olive) from various brain stem nuclei which determine when a movement has not been executed successfully.

When a climbing fiber signal arrives, only those Purkinje cell synapses that were active during the movement will be modified. Thus for similar movements in the future (where similar groups of granule cells are active) the correct response will be *anticipated* by the cerebellum. In this way the correct reflex modulation for a particular movement is learnt.

Figure 2.15 shows some of the biochemical details of the hebbian learning mechanism (from [86]). The important point to note is that when a Purkinje synapse is activated by a parallel fiber a chemical trace (diacylglycerol) is released to mark that synapse eligible for modification. When the climbing fiber is activated and the Purkinje cell fires, only the eligible synapses are modified. The chemical trace dissipates each time the climbing fiber fires. Even when there is no training signal the climbing fiber still fires approximately once per second. Note that synaptic modification does not occur if the Purkinje cell fires without a climbing fiber signal, because then the calcium channel activation will not occur.

This behavior is copied in the model of figure 2.14, except for one difference: the training signal only modifies the synapses, it does not activate the Purkinje cell. The effect of synapse eligibility is represented in the block labeled "memory", and the weight value is accumulated in the integrator block ($\int_0^t$).

Note that other studies [88] have suggested that synaptic plasticity occurs in the granule cell synapses. This is not a widely held belief but it has been used in some computational studies as a mechanism for the self-organization of the granule cell layer [46].

2.8 Motor cortex

The highest levels of the motor control hierarchy are the motor areas of the cerebral cortex. The mechanisms used by the cortex are not very well understood, but its role can be illustrated by the behavior that results from its absence. For example, removal of the cerebral cortex in cats impairs only certain types of motor function [40]. It does not interfere with the animal's ability to walk, eat, fight, and avoid obstacles in its path—these are all functions controlled at a lower level. But the animal lacks purposefulness in its movement, and it will sit very still for hours at a time. Thus the cerebral cortex adds a voluntary component to behaviors that would otherwise only be elicited by specific stimuli.

In general, the higher (evolutionarily speaking) an animal is, the more the higher levels of motor control subsume the functions of the lower levels. This is called the process of "encephalization". For example, decortication of the cat leaves it still able to respond adequately to its environment, but decortication of man causes a complete loss of all purposeful motion. This is because in the human brain the lower levels *rely* more on commands descending from the higher levels.

The *primary motor cortex* is divided into areas specific to different body parts, proportional to the amount of dexterity required in each (so the hands and face get more cortex area than the legs or feet). These areas receive input from the sensory cortex and directly from spinal cord sensors corresponding to the body part. Many of the outputs go directly to the spinal cord. This allows the motor cortex to control fine movements, which is especially useful for the hands of primates. Groups of cortical neurons within each functional region can initiate a movement of a related group of muscles to move a limb to a fixed point in space. Any one group of neurons is only active for some behaviors and not others (for example, picking up a cup of coffee but not juggling a ball), even though the behaviors may involve the same muscles. Thus the primary motor cortex is said to perform selection of motor programs.

The *pre-motor cortex* is tightly interwoven with the primary motor cortex. It is involved in the detailed planning of sequences of movements. Neurons in this area become active several hundred milliseconds before a complex movement takes place.

2.9 Biological realism

Many existing robots and automatons are biologically realistic (or biologically inspired) in some way. This is sometimes because researchers want an injection of new ideas into their designs, and sometimes because they want to model biological systems to help understand them better. For example, in [32] the body and spinal cord of the Lamprey (an eel-like fish) were simulated. It was shown that the coupled oscillators in the spinal cord could (in conjunction with sensory feedback) produce the correctly timed muscle contractions necessary for swimming. Other Lamprey studies [90, 91] explored brain stem control models and learning schemes to acquire the appropriate CPG parameters for correct swimming. Other examples are [31] and [11] which model insect walking in a neural network simulation which controls a six legged robot. Realistic inter-leg coordination mechanisms are used and it is shown that interactions between the controlling network, the robot and the environment are important.

There have been many studies of legged locomotion driven by networks of coupled oscillators. For example, [120] investigates a simple network of six neural oscillators for driving biped locomotion. With careful adjustment of the coupling parameters, simulated gaits (walking and running) can be produced on an idealized 2D mechanical model. [26] shows how CPGs made up of coupled oscillators can produce outputs for quadrupedal gaits (walking, trotting and bounding). A driving signal can be varied to generate transitions between the different gaits. [27] uses six phase-locked oscillators to drive the legs of a hexapod robot. It is shown how transitions between gaits are actually symmetry breaking bifurcations

in the coupled system dynamics. [66] has used genetic algorithms to synthesize gait-producing pattern generators in a hexapod robot.

Several groups have attempted to model the brain at a much higher level. For example, [46] describes a quasi-realistic cerebellum model used to control a robot manipulator. It contains a self organizing granule cell layer and a Purkinje layer which uses Hebb learning rules. The "Darwin" system [105] is an ambitious attempt to create a complete artificial brain for various automatons. Based on Edelman's theory of neuronal group selection, it has realistic cell and synaptic modification dynamics, and various realistic sensory and motor systems. The Darwin-III system contains 50 interconnected networks with some 50,000 cells and 620,000 synaptic junctions.

Many authors have created design paradigms based on biological principles. Crawford [30, 29] suggests a hierarchical controller using radial basis function networks for systems with many degrees of freedom, made up of a network of the simple single-joint controllers. This approach was used to control a simulated human platform diver. Altman [4] presents a distributed decision making model for insects, based on a neural equivalent of Brooks' subsumption architecture model. Kalveram [57] suggests that robot arm movements can be controlled by CPGs and reflex-like processes which allow high level centers to specify only the kinematics (not the dynamics) of movement. Hallam [41] gives a neuroethological approach for controlling a mobile robot using a neural network with quasi-realistic synapse modification.

2.10 Conclusion

The basic principles of neural control have been described, and it has been shown that the brain is a complex neural machine with many specialized components. The most fundamental behaviors are implemented by the spinal cord. These include the stretch reflex and its associated reflexes, which act as a servo mechanism to regulate muscle length. A model of this servo mechanism is used to achieve low level robot joint control in Chapter 7.

At a higher level the brain stem implements more complex behaviors, which operate through modulation of the lower levels. This "modulation principle" has been found by the author and others to be a useful guideline when designing intelligent controllers.

The cerebellum is the major adaptive component for learned motor behavior. The CMAC, which is described in the next chapter, is based on the cerebellar model of figure 2.14, although biological modeling is sacrificed to engineering implementation details to achieve a practical neural network. The FOX controller, described in Chapter 5, is an even better cerebellar model as its weights have associated eligibility values which are similar (in some respects) to the chemical eligibility trace of the cerebellar Purkinje cells.

Chapter 3

The CMAC neural network

3.1 Introduction

This chapter describes the operation of the CMAC neural network. It is largely a tutorial, although the CMAC's performance will be analyzed in detail and some new results will be presented. CMAC is an acronym for Cerebellar Model Articulation Controller[1]. The CMAC was first described by Albus in 1975 [3, 2] as a simple model of the cortex of the cerebellum (see Chapter 2). Since then it has been in and out of fashion, extended in many different ways, and used in a wide range of different applications. Despite its biological relevance, the main reason for using the CMAC is that it operates very fast, which makes it suitable for real-time adaptive control.

The operation of the CMAC will be described in two ways: as a neural network and as a lookup table. Then the properties of the CMAC will be explored. Comparisons will be drawn between the CMAC and the multi-layer perceptron (MLP) neural network, which is described in Appendix C. Other information about CMAC operation can be found in [85] and [123].

3.2 The CMAC

3.2.1 The CMAC as a neural network

The basic operation of a two-input two-output CMAC network is illustrated in figure 3.1a. It has three layers, labeled L1, L2, L3 in the figure. The inputs are the values y_1 and y_2. Layer 1 contains an array of "feature detecting" neurons z_{ij} for each input y_i. Each of these outputs one for inputs in a limited range, otherwise they output zero (figure 3.1b). For any input y_i a fixed number of neurons (n_a) in each layer 1 array will be activated ($n_a = 5$ in the example). The layer 1 neurons effectively quantize the inputs.

Layer 2 contains n_v association neurons a_{ij} which are connected to one neuron from each layer 1 input array (z_{1i}, z_{2j}). Each layer 2 neuron outputs 1.0 when all its inputs are nonzero, otherwise it outputs zero—these neurons compute the logical AND of their inputs. They are arranged so exactly n_a are activated by any input (5 in the example).

Layer 3 contains the n_x output neurons, each of which computes a weighted sum of all layer 2

[1] CMAC is pronounced "see-mac".

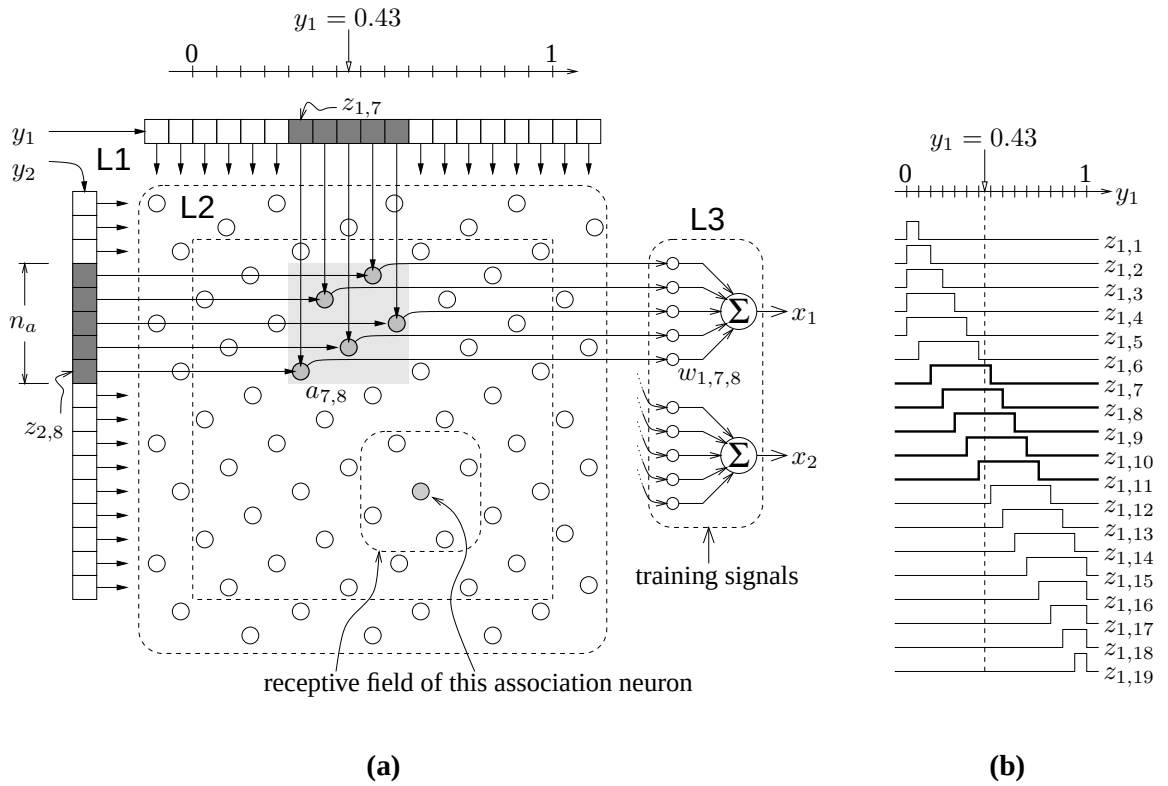


Figure 3.1: ***(a)*** *An example two-input two-output CMAC, in neural network form ($n_y = 2$, $n_a = 5$, $n_v = 72$, $n_x = 2$).* ***(b)*** *Responses of the feature detecting neurons for input 1.*

outputs, i.e.:

$$x_i = \sum_{jk} w_{ijk}\, a_{jk} \tag{3.1}$$

The parameters w_{ijk} are the weights which parameterize the CMAC mapping (w_{ijk} connects a_{jk} to output i). There are n_x weights for every layer 2 association neuron, which makes $n_v n_x$ weights in total.

Only a fraction of all the possible association neurons are used. They are distributed in a pattern which conserves weight parameters without degrading the local generalization properties too much (this will be explained later). Each layer 2 neuron has a receptive field that is $n_a \times n_a$ units in size, i.e. this is the size of the input space region that activates the neuron.

The CMAC was intended by Albus to be a simple model of the cerebellum. Its three layers correspond to sensory feature-detecting neurons, granule cells and Purkinje cells respectively, the last two cell types being dominant in the cortex of the cerebellum. This similarity is rather superficial (as many biological properties of the cerebellum are not modeled) therefore it is not the main reason for using the CMAC in a biologically based control approach. The CMAC's implementation speed is a far more useful property.

It is apparent that, conceptually at least, the CMAC deserves to be called a neural network. However, actual software implementations are much more convenient if the CMAC is viewed as a lookup table.

3.2.2 The CMAC as a lookup table

Figure 3.2 shows an alternative table based CMAC which is exactly equivalent to the above neural network version. The inputs y_1 and y_2 are first scaled and quantized to integers q_1 and q_2. In this example:

$$q_1 = \lfloor 15\, y_1 \rfloor \tag{3.2}$$
$$q_2 = \lfloor 15\, y_2 \rfloor \tag{3.3}$$

as there are 15 input quantization steps between 0 and 1 (note $\lfloor \cdot \rfloor$ is the floor function). The indexes (q_1, q_2) are used to look up weights in n_a two-dimensional lookup tables (recall that $n_a = 5$ is the number of association neurons activated for any input). Each lookup table is called an "association unit" (AU)—they are labeled AU1...AU5 in figure 3.2. The AU tables store one weight value in each cell. Each AU has cells which are n_a times larger than the input quantization cell size, and are also displaced along each axis by some constant. The displacement for AU i along input y_j is d_j^i (the example values for d_j^i are given in figure 3.2). If p_j^i is the table index for AU i along axis y_j then, in the example:

$$p_j^i = \left\lfloor \frac{q_j + d_j^i}{n_a} \right\rfloor = \left\lfloor \frac{q_j + d_j^i}{5} \right\rfloor \tag{3.4}$$

The CMAC mapping algorithm (based on the table form) is shown in table 3.1. Software CMAC implementations never use the neural network form. The CMACHASH function used by the CMACQUANTIZEANDASSOCIATE procedure takes the AU number and the table indexes and generates an index into a weight array. Implementations of CMACHASH, and the way the weights are arranged in memory, will be described later.

The CMAC is faster than an "equivalent" MLP network because only n_a neurons (i.e. n_a table entries) need to be considered to compute each output, whereas the MLP must perform calculations for all of its neurons.

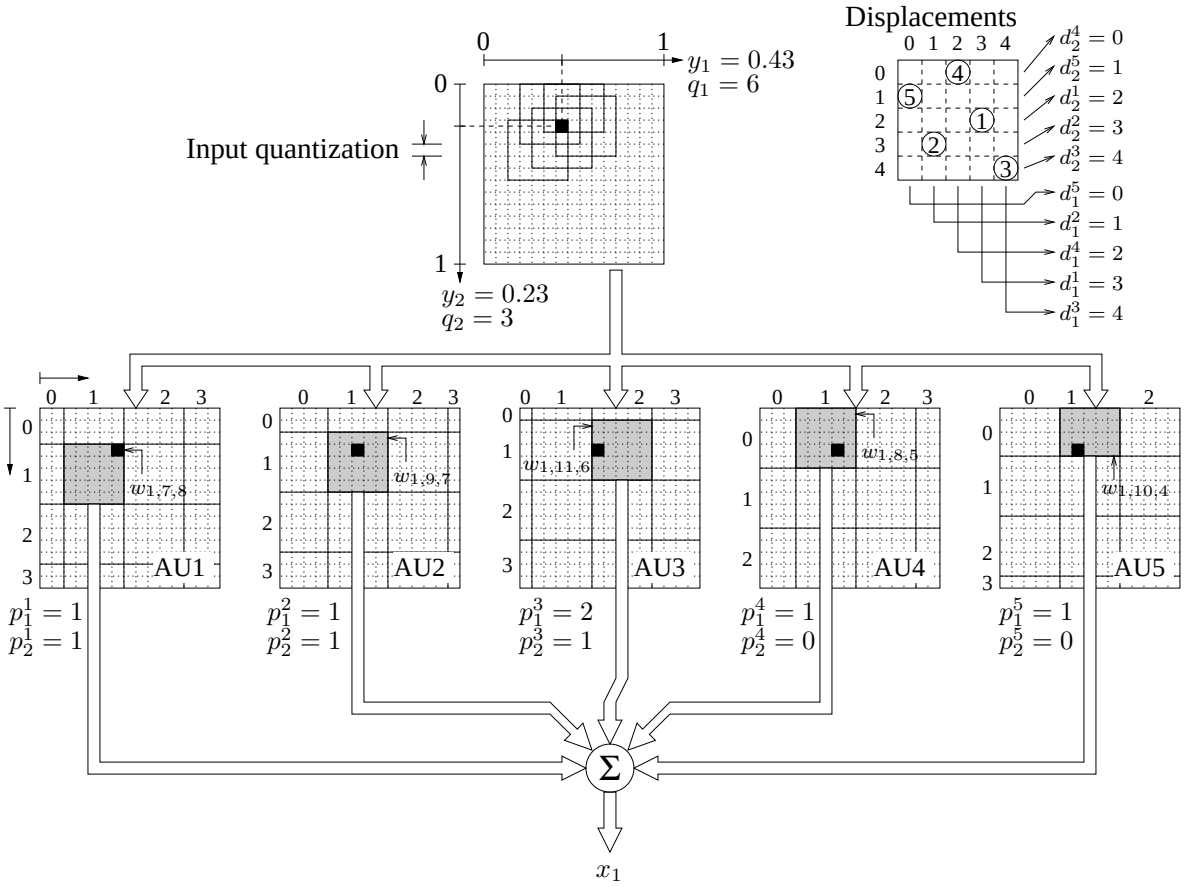


Figure 3.2: *A two input CMAC (table form) equivalent to figure 3.1,* $n_a = 5$.

THE CMAC ALGORITHM

Parameters

n_y — Number of inputs (integer ≥ 1)
n_x — Number of outputs (integer ≥ 1)
n_a — Number of association units (integer ≥ 1)
d_j^i — The displacement for AU i along input y_j (integer, $0 \leq d_j^i < n_a$)
min_i and max_i — Input ranges; the minimum and maximum values for y_i (real).
res_i — Input resolution; the number of quantization steps for input y_i (integer > 1).
n_w — The total number of physical CMAC weights per output.

Internal state variables

μ_i — An index into the weight tables for association unit i ($i = 1 \ldots n_a$).
$W_j[i]$ — Weight i in the weight table for output x_j, $i = 0 \ldots (n_w - 1)$, $j = 1 \ldots n_x$.

ALGORITHM: CMACMAP — Maps the CMAC input to its output.
Inputs: $y_1 \ldots y_{n_y}$ (scalars)
Outputs: $x_1 \ldots x_{n_x}$ (scalars)

$\Rightarrow$ CMACQUANTIZEANDASSOCIATE $(y_1 \ldots y_{n_y})$ —this sets $\mu_1 \ldots \mu_{n_a}$
for $i \leftarrow 1 \ldots n_x$ —for all outputs
$x_i \leftarrow 0$
for $j \leftarrow 1 \ldots n_a$: $x_i \leftarrow x_i + W_i\,[\mu_j]$ —for all AUs add table entry to x_i

---

ALGORITHM: CMACQUANTIZEANDASSOCIATE — Get weight table indexes.
Inputs: $y_1 \ldots y_{n_y}$ (scalars)
Outputs: $\mu_1 \ldots \mu_{n_a}$ (scalars)

$\Rightarrow$ for $i \leftarrow 1 \ldots n_y$ —for all inputs
if $y_i < \text{min}_i$ then : $y_i \leftarrow \text{min}_i$ —limit y_i
if $y_i > \text{max}_i$ then : $y_i \leftarrow \text{max}_i$ —limit y_i
$q_i \leftarrow \left\lfloor \text{res}_i \dfrac{y_i - \text{min}_i}{\text{max}_i - \text{min}_i} \right\rfloor$ —quantize input
if $q_i \geq \text{res}_i$ then : $q_i \leftarrow \text{res}_i - 1$ —enforce $0 \leq q_i < \text{res}_i$
for $i \leftarrow 1 \ldots n_a$ —for all AUs
for $j \leftarrow 1 \ldots n_y$: $p_j^i \leftarrow \left\lfloor \dfrac{q_j + d_j^i}{n_a} \right\rfloor$ —find tables indexes for all inputs
$\mu_i \leftarrow$ CMACHASH $(i, p_1^i, p_2^i, \ldots, p_{n_y}^i)$ —Get weight index for AU i

---

ALGORITHM: CMACTARGETTRAIN — Train the CMAC output to reach a given target.
(It is assumed that CMACMAP has already been called.)
Inputs: $t_1 \ldots t_{n_x}$, α (scalars)

$\Rightarrow$ for $i \leftarrow 1 \ldots n_x$
increment $\leftarrow \alpha \dfrac{t_i - x_i}{n_a}$
for $j \leftarrow 1 \ldots n_a$: $W_i\,[\mu_j] \leftarrow W_i\,[\mu_j] + \text{increment}$

Table 3.1: *The algorithms for computing the CMAC output and training the CMAC.*

3.3 Training

The training process takes a set of desired input to output mappings (training points) and adjusts the weights so that the global mapping fits the set. It will be useful to distinguish between two different CMAC training modes:

- **Target training**: First an input is presented to the CMAC and the output is computed. Then each of the n_a referenced weights (one per lookup table) is increased so that the outputs x_i come closer to the desired target values t_i, i.e.:

$$w_{ijk} \leftarrow w_{ijk} + \frac{\alpha\, a_{jk}}{n_a}(t_i - x_i) \tag{3.5}$$

  where α is the learning rate constant $(0 \leq \alpha \leq 1)$. If $\alpha = 0$ then the outputs will not change. If $\alpha = 1$ then the outputs will be set equal to the targets. This is implemented in the CMACTARGETTRAIN procedure shown in table 3.1.

- **Error training**: First an input is presented to the CMAC and the output is computed. Then each of the n_a referenced weights is incremented so that the output vector $[x_1 \ldots x_{n_x}]$ increases in the direction of the error vector $[e_1 \ldots e_{n_x}]$, i.e.:

$$w_{ijk} \leftarrow w_{ijk} + \frac{\alpha\, a_{jk}}{n_a} e_i \tag{3.6}$$

  This results in a trivial change to the CMACTARGETTRAIN procedure given in table 3.1.

Target training causes the CMAC output to seek a given *target*, error training causes it to grow in a given *direction*. The input/desired-output pairs (training points) are normally presented to the CMAC in one of two ways:

- **Random order**: If the CMAC is to be trained to represent some function that is known beforehand, then a selection of training points from the function can be presented in random order to minimize learning interference [122].

- **Trajectory order**: If the CMAC is being used in an online controller then the training point inputs will probably vary gradually, as they are tied to the system sensors. In this case each training point will be close to the previous one in the input space.

3.4 Hashing

3.4.1 The number of CMAC weights

How are the weights in the weight tables stored in the computer's memory? The naïve approach, storing each weight as an separate number in a large floating point array, is usually not possible. To see why, consider the number of weights required for the CMAC parameters given in table 3.1. The maximum value of the table index p_j^i is (if the largest displacements are assumed):

$$\text{maximum } p_j^i = \left\lfloor \frac{(\text{maximum } q_j) + (\text{maximum } d_j^i)}{n_a} \right\rfloor \tag{3.7}$$

$$= \left\lfloor \frac{(\text{res}_j - 1) + (n_a - 1)}{n_a} \right\rfloor \tag{3.8}$$

$$= \left\lfloor \frac{\text{res}_j - 2}{n_a} \right\rfloor + 1 \tag{3.9}$$

The total number of CMAC weights is the number of AU tables times the number of weights stored per AU:

$$\text{weights in CMAC} = n_a \times \prod_{j=1}^{n_y} \left\{ \text{maximum } p_j^i + 1 \right\} \tag{3.10}$$

$$= n_a \times \prod_{j=1}^{n_y} \left\{ \left\lfloor \frac{\text{res}_j - 2}{n_a} \right\rfloor + 2 \right\} \tag{3.11}$$

To simplify this, assume that res_j is sufficiently large and the same for all j, so p_j^i can be approximated by

$$\text{maximum } p_j^i \approx \frac{\text{res}}{n_a} \tag{3.12}$$

which gives

$$\text{weights in CMAC} \approx n_a \left(\frac{\text{res}}{n_a} \right)^{n_y} \tag{3.13}$$

$$= \frac{\text{res}^{(n_y)}}{{n_a}^{(n_y-1)}} \tag{3.14}$$

There are two things to note about equation 3.14. First, increasing n_a *reduces* the number of weights, even though it increases the number of weight tables. This fact will be used later to try to get a smoother output from the CMAC.

The second thing to note is that, for typical values of res, n_a and n_y the number of weights is huge. For example, suppose res $= 200$, $n_a = 10$ and $n_y = 6$ (these values are not atypical) then the number of weights is 640 million. At four bytes per floating point number this would require 2441 megabytes of memory, which is presently unreasonable. The problem is that the number of weights is exponential in the number of input dimensions n_y. One way to reduce the number of weights is to use hashing.

3.4.2 What is hashing?

Hashing is a commonly used technique in computer science [109]. The idea is to take an address into a large "virtual" memory and map it into an address into a smaller physical memory. For example, suppose a program is required to quickly store and recall 100 seven digit telephone numbers[2]. In would be wasteful and inefficient to use the telephone number as an index into an array of 10^7 entries. Instead an array of only (say) 251 entries[3] could be used, and the index into this array could be computed with

$$\text{index} = (\text{phone number}) \bmod 251 \tag{3.15}$$

The modulo function hashes the large phone number address into a smaller index in the range $0 \ldots 250$. This makes more efficient use of memory, but now there is the problem that some of the phone numbers will hash to the same physical memory location (a problem that gets worse as more numbers that have to be stored). This is known as "hash collision". A realistic database program would have to resolve hash collision using collision lists or secondary probing techniques [109].

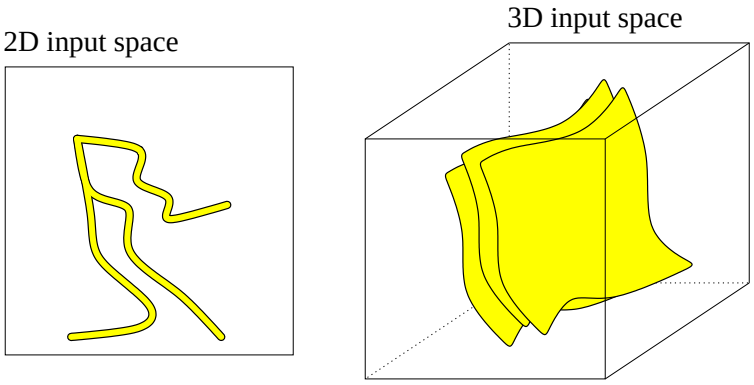


Figure 3.3: *An illustration of how CMAC inputs derived from control processes can tend to fall along low dimensional surfaces (one and two dimensional examples).*

3.4.3 How the CMAC uses hashing

The CMAC uses a hashing scheme to map the large "virtual" weight table address into a smaller physical one. Hash collisions are ignored—if any two weights with the same physical address are written, the second one will simply replace the first. Consider using a 100 kilobyte physical memory for the example above which has a 2441 megabyte virtual memory requirement. In this case almost 25000 virtual weights would map to a single physical weight, which would seem to hopelessly compromise any training that was performed.

In practice this is not a problem if the inputs to a CMAC only venture over a fraction of the entire input space volume, because then only a subset of the virtual weights will be referenced. This is usually true for inputs that come from physical sensors attached to some controlled process. Sensor values may be correlated with one another, and the process may only be observed in some fixed modes of operation (or in limit cycles), which means that only a small subset of all possible input trajectories are seen. This usually means that the CMAC inputs only lie on a few low dimensional surfaces in the input space (figure 3.3).

The hash collision problem is intolerable if the inputs roam over the *entire* input space. But if the CMAC input parameters are correctly chosen then it will be tolerable over the set of *likely* inputs. Thus the benefit of hashing is that it allows the relatively small number of physical weights to be allocated *where they are needed* in the input space.

Hash collisions are manifested as noise in the CMAC output. As a larger volume of the input space is used, more hash collisions will be experienced, the CMAC output will get noisier and it will be harder to get the CMAC to retain the training data.

3.4.4 CMAC hashing algorithms

A CMAC hashing algorithm (CMACHASH) generates a physical weight table index i from the AU number j and the tables indexes $p_1^j, p_2^j, \ldots, p_{n_y}^j$ (from table 3.1), i.e.

$$i \quad = \quad f_h\Big(f_v(p_1^j, p_2^j, \ldots, p_{n_y}^j), j\Big) \qquad \text{—variant 1} \tag{3.16}$$

[2] Any programmer can think of many ways to tackle this problem (e.g. linked lists and binary trees) but only hashing is interesting here.

[3] 251 is a prime number. Prime numbers have been shown to be particularly good with the modulo hash function [109].

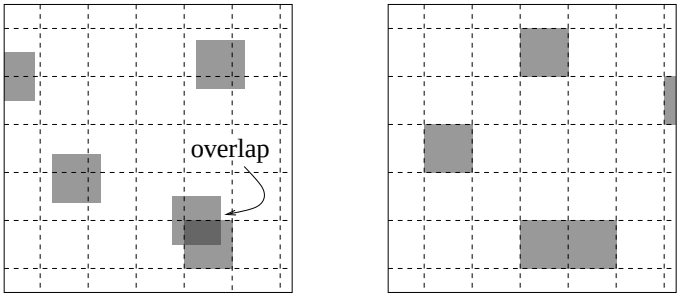


Figure 3.4: *When hashing is used, a single physical weight influences multiple virtual weights, resulting in disjointed "basis functions" — (a) shows an example of variant 1 hashing (AU number hashed) and (b) shows an example of variant 2 (AU number not hashed).*

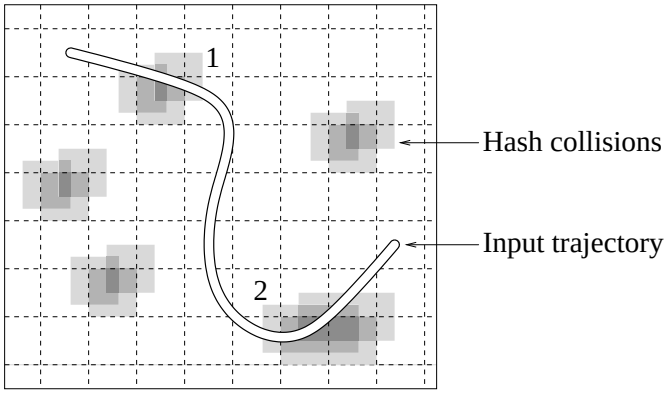


Figure 3.5: *This is what happens when variant-2 hashing does not hash on the AU index (the trajectory intersects two hash-collision clusters).*

$$\text{or} \qquad i \;=\; f_h\Big(f_v(p_1^j, p_2^j, \ldots, p_{n_y}^j), j\Big) + n_p\, j \qquad \text{—variant 2} \tag{3.17}$$

where f_v is a function that generates a unique virtual address $(0 \ldots n_v - 1)$ from the weight table indexes, and f_h is a hashing function that produces a physical address $(0 \ldots n_p - 1)$. A single physical weight will control multiple virtual weights, as shown in figure 3.4—this set is called the basis function for that weight[4]. Figure 3.4a shows how variant 1 hashes on the AU index, so a basis function can have weights in multiple AUs. Thus the basis function can have overlapping AU cells. Figure 3.4b shows how variant 2 uses the AU index j to index a separate n_p-sized area for each AU, ensuring that each basis function contains only cells from a single AU.

Variant 2 must also hash on the AU index j, because otherwise the basis functions for corresponding weights in each AU will have the same pattern. Consider figure 3.5, which shows what can happen when variant 2 hashing does not include the AU index. Basis functions localized in more than one useful area of the input space are clustered together, so training performed at point 1 will cause points 1 and 2 to change by the same amount. If the AU index is included in the hash function then the hash collisions will

[4]This terminology is not strictly correct, mathematically speaking.

be widely distributed, so the training effect at point 2 will have only $1/n_a$ of the full effect. Either variant will work, but it will be seen later that variant 2 makes some aspects of the FOX algorithm simpler.

There are numerous choices for f_v and f_h. The major requirements are that f_v should produce a different index for every virtual weight, and f_h should not result in a physical weight having some systematic pattern of virtual weights that will lead to greater than expected hash collisions (i.e. the more "random" the hash function is the better).

Two examples of f_v are shown in table 3.2, one suited for software implementation and the other for hardware. Two examples of f_h are shown in figure 3.6.

3.5 Properties of the CMAC

3.5.1 Limited input space

Each CMAC input has a minimum and maximum value, beyond which the weight tables do not hold any weight values. Thus the designer needs to know beforehand what the probable input ranges are. The mapping algorithm in table 3.1 clamps the inputs to be between the minimum and maximum values. The CMAC will not perform as expected if the input goes outside this valid range, but valid weight table indexes will still be generated.

3.5.2 Piecewise constant

The CMAC input to output mapping is piecewise constant because of the quantized input. In other words the mapping contains many discontinuous steps. This may seem likely to result in a poor mapping, but in fact it is not a great disadvantage in many applications. If the input resolutions are chosen to be large enough then the discontinuities will be small. There is a trade off, however, because more input resolution results in a larger virtual weight space.

3.5.3 Local generalization

The concept of generalization is explained in Appendix A. In contrast to the multi-layer perceptron (MLP), the CMAC has *local* generalization (LG). This means that when each data point is presented to the CMAC training algorithm, only a small region of the mapping around that point is adjusted. This occurs, of course, because only the weights which affect the output are adjusted, and each of those weights can only influence a small area of the mapping. Hashing increases the area that each physical weight influences, but this does not count as generalization as it is unpredictable. Figure 3.7 shows a possible result of one CMAC training iteration for $\alpha = 1$.

Local generalization can also be regarded as interpolation. The interpolation capabilities of the binary CMAC are explored further in [22].

3.5.4 Training sparsity

To ensure good generalization, how far apart (in the input space) can the training points be in trajectory-ordered data? Figure 3.8 shows the CMAC mapping that results from training with sparse data, ranging from 6 to 21 training points spaced evenly between zero and one. In each case the training was performed on each training point in turn with a learning rate $\alpha = 0.1$, repeating this until convergence was achieved. This CMAC had 100 quantization steps in the [0,1] interval and $n_a = 10$, so the LG area had 20% of the [0,1] interval width.

ALGORITHMS TO COMPUTE VIRTUAL ADDRESS (f_v)

Purpose

Compute a virtual address for weight table indexes $p_1^j, p_2^j, \ldots, p_{n_y}^j$.

ALGORITHM 1 (software)

Definitions

$r_k = \left\lfloor \dfrac{\text{res}_k - 2}{n_a} \right\rfloor + 2$ — the number of different values of p_k^j.

Inputs: $p_1^j, p_2^j, \ldots, p_{n_y}^j$.

Output: h—A virtual address in the range $0 \ldots \left(\prod_{k=1}^{n_y} r_k \right) - 1$

$\Rightarrow$ $h \leftarrow 0$

for $i \leftarrow 1 \ldots n_y$

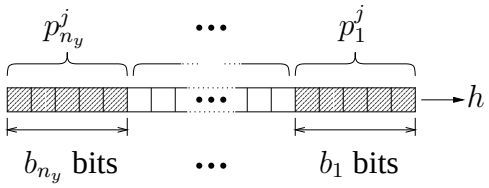
$h \leftarrow h\, r_i + p_i^j$

ALGORITHM 2 (hardware)

Inputs: $p_1^j, p_2^j, \ldots, p_{n_y}^j$.

Output: h—A virtual address.

$\Rightarrow$ Insert the indexes $p_1^j \ldots p_{n_y}^j$ directly into a hardware register:



$b_1 \ldots b_{n_y}$ are such that $2^{b_i} \geq r_i$.

Table 3.2: *Two algorithms for computing a virtual weight address from the weight table indexes.*

Here is a highly effective hash function in C, adapted from [100]. It does a pseudo-DES hashing of the two 32 bit arguments to a 32 bit result. The argument `virtual` *is the virtual address returned by* f_v *(it must have a maximum value less than* 2^{32}*),* `au_num` *is the association unit number, and* `pbits` *is the number of bits to allow in the physical address (*$0 \ldots 31$*). This does a much better job of nonlinear pseudo-random bit mixing than some other hash functions (like mod-prime or linear congruential) but at the expense of speed (it's still pretty quick, though).*

```
unsigned long pdes_hash (unsigned long virtual, unsigned long au_num,
                         int pbits)
{
  unsigned long i,ia,ib,iswap,itmph=0,itmpl=0;
  static unsigned long c1 [4] = {
    0xbaa96887L,0x1e17d32cL,0x03bcdc3cL,0x0f33d1b2L};
  static unsigned long c2 [4] = {
    0x4b0f3b58L,0xe874f0c3L,0x6955c5a6L,0x55a7ca46L};
  /* Perform 4 iterations of DES logic, using a simpler       */
  /* (non-cryptographic) nonlinear function instead of DES's. */
  for (i=0; i<4; i++) {
    ia = (iswap=au_num)^c1[i];
    itmpl = ia & 0xffff;
    itmph = ia >> 16;
    ib = itmpl*itmpl + ~(itmph*itmph);
    au_num = virtual^(((ia=(ib>>16)|((ib&0xffff)<<16))^c2[i])+
             itmpl*itmph);
    virtual = iswap;
  }
  /* Chop off the lower 'pbits' bits of the hashed value */
  return virtual & ((1 << pbits)-1);
}
```

This hash function is particularly useful for hardware CMAC implementations. The weight table indexes and the AU number j *are loaded into a linear feedback shift register [100] which is then shifted sufficiently to scramble the bits. The physical address is taken as some subset of the shift register bits. Enough shifts must be performed so that each original bit affects a number of bits in the physical address. See [54] for further information.*

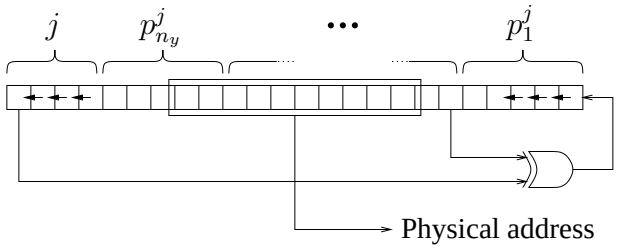


Figure 3.6: *Some hashing algorithms.*

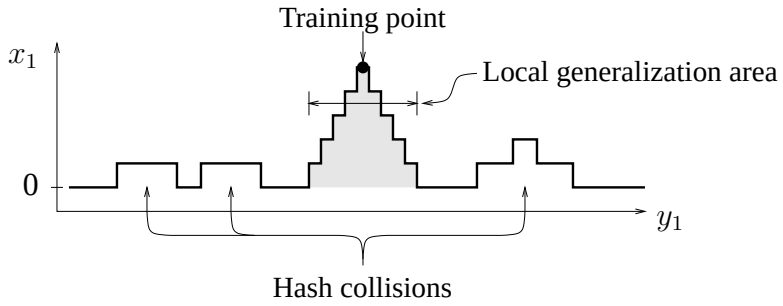


Figure 3.7: *An example CMAC output after one training iteration starting from zero ($n_a = 5$).*

For points where the LG regions don't overlap (the graphs of 5 and 6 points) the mapping is a series of triangular bumps. As the points get closer together (7,8,9,10 points) the regions between them become better interpolated, until at 11 points the LG regions overlap exactly and the inter-point spaces are linearly interpolated—this is a reasonably good mode of generalization. As the training points get closer still (12,13,. . .) the inter-point spaces are no longer linearly interpolated, but the error there remains relatively small. Linear interpolation is achieved again at 21 points. Obviously, better generalization is achieved if the training points are closer together.

The bottom three graphs in figure 3.8 show the same CMAC after training with 10, 100 and 1000 randomly positioned points in the interval [0,1] ($\alpha = 0.5$). 10 points is too few to get a good approximation with $\alpha < 1$. The approximation becomes smoother with more training points because the points tend to be closer together. Note the region on the left of the 100 point graph where the lack of training points causes a small interval of large error.

In conclusion, for a small number of fixed training points the generalization performance is inferior to the MLP, as the inter-point regions are hardly ever perfectly interpolated. As the inter-point space decreases below the size of the LG region, the generalization becomes better. For a large number of "randomly" spaced training points (perhaps generated by some function or process) the CMAC is a good functional approximator.

3.5.5 Training interference

Consider training a CMAC at two points, first A then B, with some learning rate α. If point B is within the LG area of point A it can make the mapping error at point A worse. This effect is called training interference.

Figure 3.9 shows the cause of the problem. If points A and B have overlapping LG areas then they share some weights (v weights in the figure). Training at B will affect the value of x_A because v of its n_a weights will be altered. If the "training interference" is defined to be the degree of influence over x_A that training at B has then:

$$\text{training interference (\%)} \quad = \quad \frac{v}{n_a} \times 100\% \tag{3.18}$$

If $v = 0$ then there is obviously no interference (because there is no overlap) and if $v = n_a$ then A and B are the same point and the interference is maximized. This can be particularly problematic during trajectory training, because then successive training points are very likely to be close to each other. The

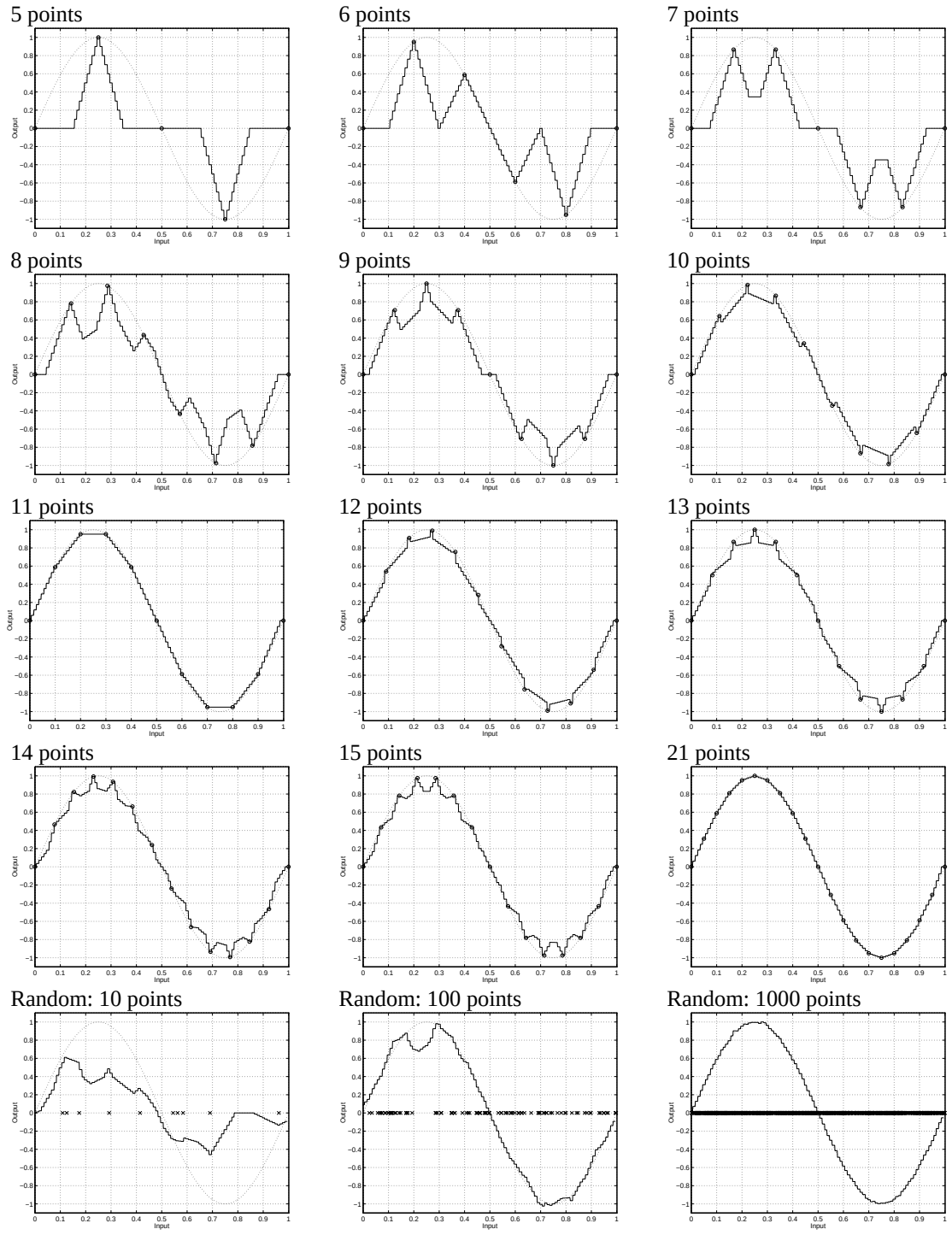


Figure 3.8: *Top four rows: the CMAC mapping resulting from training with sparse data, ranging from 6 to 21 training points spaced evenly between zero and one (marked with $\circ$'s). Bottom row: the CMAC mapping resulting from training with 10, 100 and 1000 randomly positioned points (input positions marked with $\times$'s, $\alpha = 0.5$). In both cases there are 100 quantization steps in the [0,1] interval, $n_a = 10$ and the training target for input y is $\sin 2\pi y$.*

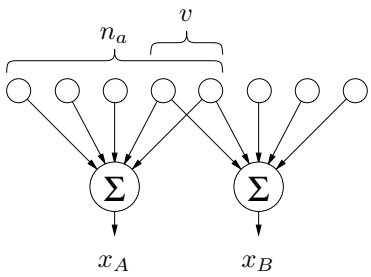


Figure 3.9: *The CMAC outputs for two different inputs that are within each other's local generalization areas.*

problem can be reduced with a lower learning rate. This effect is further explored in [122] for different training techniques.

3.5.6 Multidimensional inflexibility

A single input CMAC has enough weight parameters so that it can generate an independent output for each quantized input. In standard CMACs with more than one input this can not be done, because (as shown in figure 3.1) the association neurons (or weight table entries) are distributed rather sparsely in the quantized input space.

However, it was said above that for n_a weight tables, the table cell size was n_a times as coarse as the quantized input cells—but this is merely a convention, not a requirement, and so there could be n_T weight tables with a cell size that is n_c times coarser than the input cells. These tables would still have to be displaced from each other (because if two tables are perfectly aligned then one is redundant). The pattern of displacement is given in a displacement diagram (an example is shown in figure 3.2). Figure 3.10 shows some other displacement diagrams, along with their local generalization areas and association neuron layouts. In practice, $n_c = n_T = n_a$ is almost always used.

Figure 3.10c is fully populated, and so it is capable of representing a different output for each input cell, but normally this flexibility is not required. To reduce the number of weights, sparser distributions like figure 3.10a or figure 3.10b are commonly used. The displacement table should have at least one entry for each row and column. Figure 3.10b shows the simplest arrangement ($d_j^i = i$) which was suggested by Albus [2] and has been commonly used.

3.5.7 Comparison with the MLP

To represent a given set of training data the CMAC normally requires more parameters than an MLP. The CMAC is often over-parameterized for the training data, so the MLP has a smoother representation which ignores noise and corresponds better to the data's underlying structure. The number of CMAC weights required to represent a given training set is proportional to the volume spanned by that data in the input space.

The CMAC can be more easily trained on-line than the MLP, for two reasons. First, the training iterations are fast. Second, the rate of convergence to the training data can be made as high as required, up to the instant convergence of $\alpha = 1$ (although such high learning rates are never used because of training interference and noise problems). The MLP always requires many iterations to converge, regardless of its learning rate.

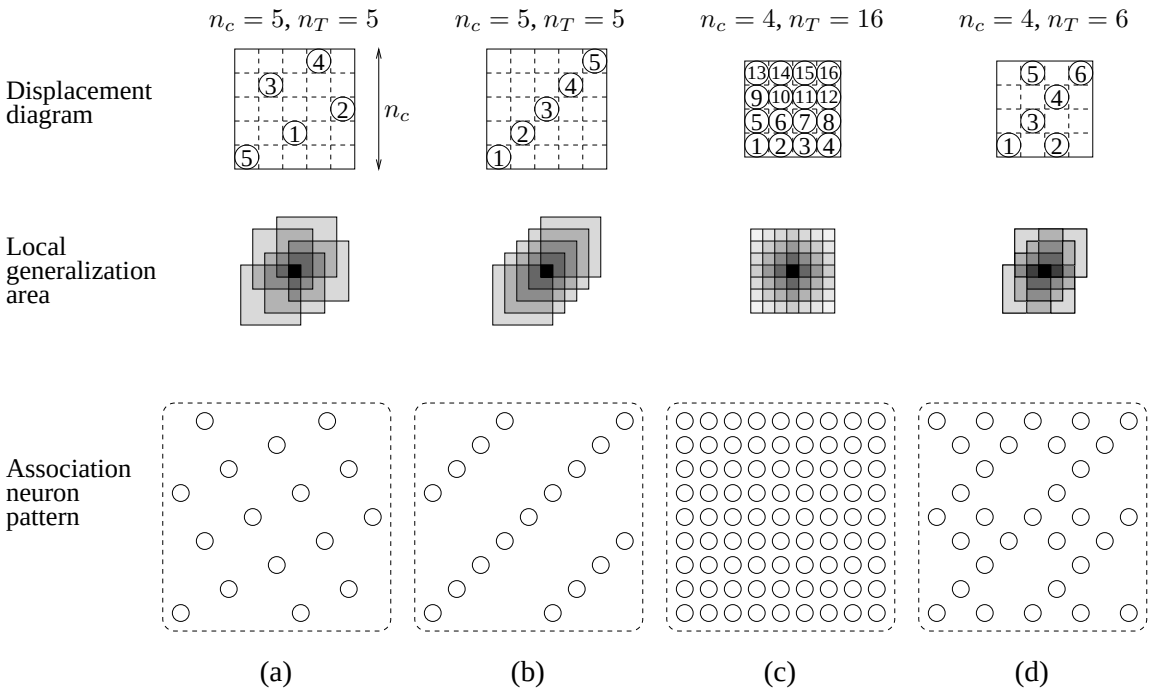


Figure 3.10: *Some example CMAC displacement diagrams and how they affect the local generalization area and association neuron layout.*

3.6 Design decisions

3.6.1 Input issues

When training with trajectory data, the input resolutions and n_a should be chosen to ensure that there is no training sparsity, i.e. so that successive training points are within the LG area of each other.

It can be useful to apply some function to an input before presenting it to the CMAC. These functions normally compress or expand the input space in certain areas so that the mapping detail can be concentrated where it is needed. A common choice is some variant of the sigmoid function

$$f(x) \quad = \quad \frac{1}{1 + e^{-x}} \tag{3.19}$$

3.6.2 Overlay displacements

There are two common choices when choosing the overlay displacement values d_j^i:

1. Choose $d_j^i = i$ which means that the AU tables are aligned along a hyperdiagonal in the input space. This is reasonably effective and particularly easy, especially for hardware CMAC implementations.

2. Choose the d_j^i to get AU clustering that is optimal in some sense. For example, Parks and Militzer ([94] and [21, appendix B]) computed overlay displacement tables for n_a up to 100 and n_y up to fifteen, such that the minimum hamming distance between any two overlay displacement vectors

is maximized. It was found that particularly good overlay displacements are achieved when n_a is prime and equal to $2n_y + 1$, and bad overlay displacements are achieved when n_a is divisible by 6.

3.6.3 Hashing performance

Hash collisions limit the CMAC mapping accuracy. This problem will now be analyzed quantitatively. Let V be the fraction of the input space hypercube that contains the set of likely CMAC inputs. Note that this includes the local generalization area around that set (so one dimensional trajectories occupy a finite volume). Once the CMAC is trained, hashing will cause the area outside V to map to random noise. Define the "noise factor" to be:

$$\text{noise factor} \triangleq \text{virtual weights mapped to each physical weight} \tag{3.20}$$

$$= \frac{V \cdot (\text{virtual weights})}{(\text{physical weights})} \tag{3.21}$$

$$= \frac{V \cdot \text{res}^{(n_y)}}{n_a^{(n_y-1)}\, n_w} \qquad \text{(from equation 3.14)} \tag{3.22}$$

The noise factor for a particular configuration limits the accuracy which the CMAC mapping can achieve. Higher noise factors mean that more hash collision noise can be expected. For example, figure 3.11 shows a single-input single-output CMAC trained on a simple target function using 10,000 random training points and $\alpha = 0.3$. The graphs show the result for four different numbers of physical weights, with less weights giving a higher noise factor and a poorer mapping.

Suppose that the input trajectory is one dimensional and of length ℓ, and that the input space hypercube has side length s. Then

$$V = \frac{\ell \cdot (\text{trajectory width})^{n_y-1}}{s^{n_y}} \tag{3.23}$$

$$= \frac{\ell \cdot \left(n_a \frac{s}{\text{res}}\right)^{n_y-1}}{s^{n_y}} \tag{3.24}$$

Which gives the following noise factor:

$$\text{noise factor} = \frac{\ell}{s} \cdot \frac{\text{res}}{n_w} \tag{3.25}$$

Note that this is independent of n_a and n_y, so for this ideal case the designer is free to choose those parameters based on other considerations.

The noise factor is not a very useful concept in practice, because V is hardly ever known beforehand—it is completely dependent on the system in which the CMAC is used. In many circumstances it is better to use a trial and error method to determine good CMAC parameters. Some reasonable CMAC parameters are initially chosen, and if the CMAC output is too noisy then the number of weights or n_a is increased, or the input resolution is reduced. Computer memory is cheap, so in many situations it is acceptable to initially use far more weights than are really needed.

3.6.4 Number of AUs

As n_a is increased, the number of parameters controlling the CMAC mapping is reduced. A single input CMAC has at least as many virtual weights as there are quantized input cells, so it is capable of representing any function. But standard multiple input CMACs do not have this property, so as n_a is

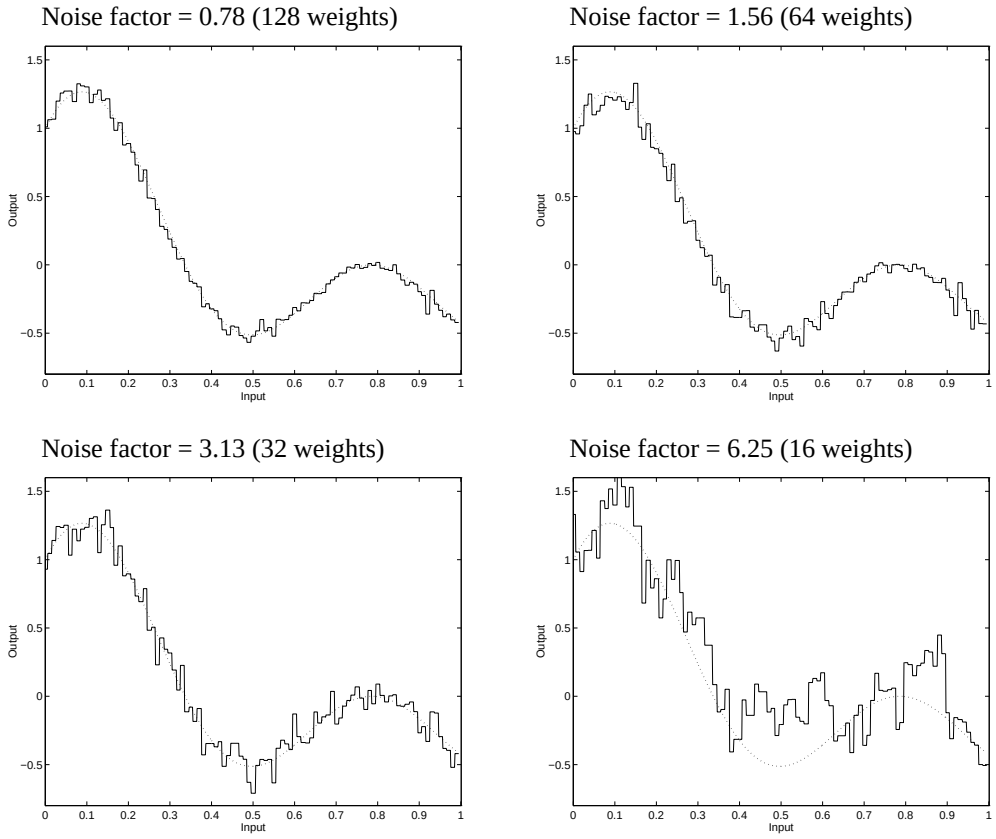


Figure 3.11: *Example CMAC mappings with different noise factors. The target function is* $x_1 = \sin 6y_1 + \cos 8y_1$, *and* $n_a = 10$.

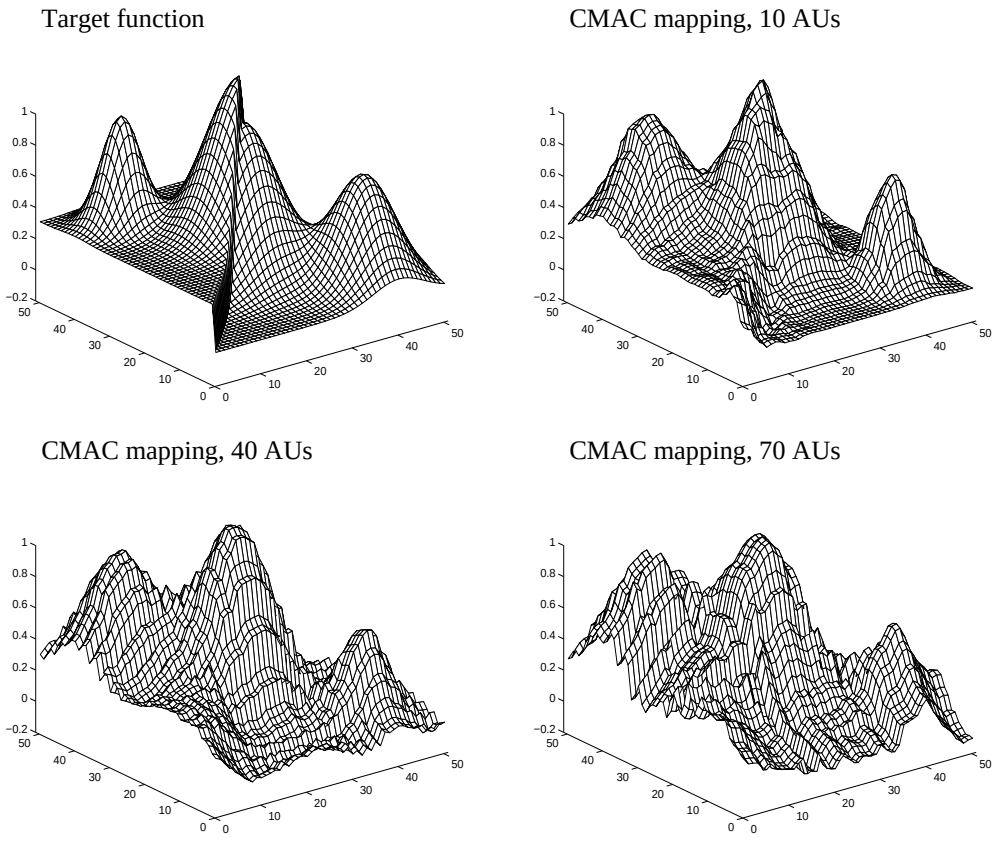


Figure 3.12: *Example 2-input CMAC mappings. The top left graph is the target function, the others show the best CMAC mappings for different AU numbers (n_a). The input resolution is 100 quantization steps along each input.*

increased the mapping flexibility is reduced. Figure 3.12 demonstrates what happens with two inputs. The top left graph is the target function—notice that it has a step discontinuity. The others graphs show CMAC mappings for n_a =10, 40 and 70 (these were trained using 10,000 random training points and $\alpha = 0.3$).

A higher n_a makes the mapping less accurate—for example it is less able to track quick variations. These inaccuracies are not the result of hash collisions (The number of weights was set to 2^{20} in this example to prevent that). With less parameters, the mapping is simply more constrained by the CMAC architecture. This can be useful to ensure a certain degree of "smoothness" in the CMAC output to compensate for an underconstraining training process. Note however that the CMAC can always perfectly represent training points along any *one-dimensional* trajectory, even for a high-dimensional CMAC.

3.6.5 Weight smoothing

As shown in figure 3.8, even if training points are closer together than the LG distance, perfect linear interpolation may not be achieved. A useful technique called "weight smoothing" was developed by the author to correct this problem, although it was never used in any application. After each training iteration the n_a indexed weights are adjusted according to

$$\text{for } j \leftarrow 1 \ldots n_a : W_i[\mu_j] \quad \leftarrow \quad (1-\alpha_n) W_i[\mu_j] + \frac{\alpha_n}{n_a} \sum_{k=1}^{n_a} W_i[\mu_k] \tag{3.26}$$

In other words, each weight is moved towards the average of all the weights by an amount proportional to α_n. Repeated training with a small value of α_n distributes the weight more evenly, reducing the gaps between the largest and smallest weights.

Figure 3.13 shows the effects. These graphs were generated the same way as in figure 3.8, but the weight smoothing training algorithm was used. The top four rows, which use $\alpha_n = 0.01$, show that points closer than the LG distance are almost perfectly linearly interpolated. The bottom row, which uses $\alpha_n = 0.2$, show some benefit as well. The 100 point graph has a smaller error than figure 3.8 in the region where the training points are sparse. But there is a slight negative side-effect: the 1000 point graph shows a slight distortion from the desired function.

3.7 Extensions

Many modifications and extensions have been made to the basic CMAC concept. For example:

- Lane, Handelman and Gelfand [63] describe "higher order" CMAC networks. In these networks the layer 1 feature detecting neuron outputs[5] respond with n'th order spline functions (e.g. triangular or cubic) instead of just a binary 0/1, although the region of nonzero activation is still bounded the same way. With careful selection of the spline parameters, the CMAC mapping becomes smooth with analytical derivatives. Higher order CMAC networks can be chained together in multi-layer and hierarchical architectures that can be trained using conventional backpropagation techniques. However, they can require more weight parameters than a binary CMAC to achieve a given level of accuracy.

- Brown and Harris [21] provide a unified description which links the CMAC to related neural and fuzzy networks, including the B-spline network. They analyze the common features of these

[5] Refer to figure 3.1b.

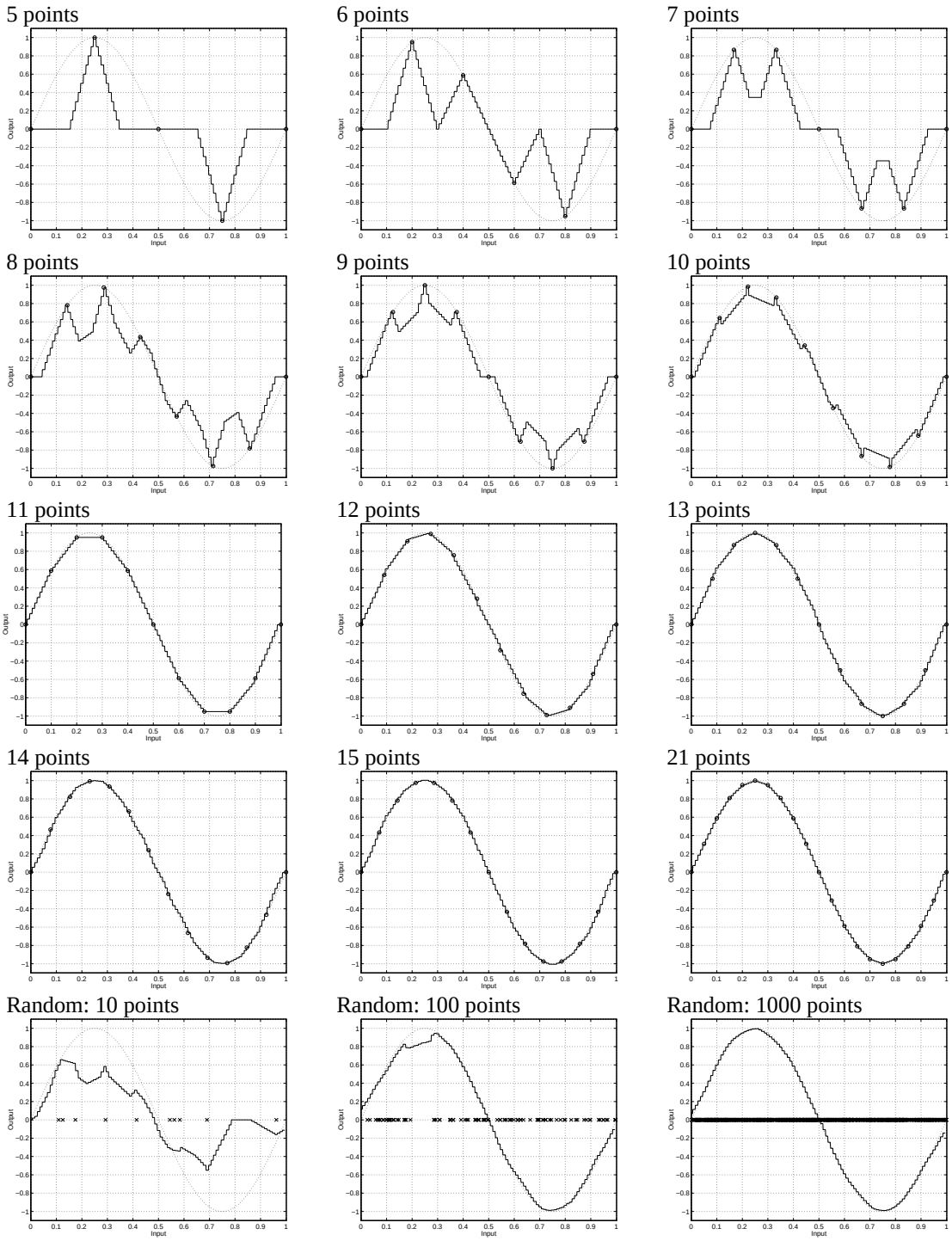


Figure 3.13: *This shows the same results as figure 3.8, but the weight smoothing training algorithm has been used. The top four rows use* $\alpha_n = 0.01$*, and the bottom row uses* $\alpha_n = 0.2$*.*

networks (such as local learning and interpolation) and compare them on a number of benchmark problems.

- Adaptive encoding of the CMAC inputs has been used to focus the representational detail on hard-to-model areas of the input space [33].
- There have been several efforts at CMAC hardware implementation, for example [54] which uses programmable logic devices. Hardware is less advantageous for the CMAC than it is for other neural network architectures, because the CMAC is so fast in software. The most difficult hardware design aspect is usually the hashing algorithm. Solutions like that in table 3.2 are typically used.

3.8 Conclusion

The CMAC neural network has the following advantages:

- The mapping and training operations are extremely fast. The time taken is proportional to the number of association units.
- The algorithms are easy to implement.
- Local generalization prevents over-training in one area of the input space from degrading the mapping in another (unless there are too few physical weights).

It has the following disadvantages:

- Many more weight parameters are needed than for, say, the multi-layer perceptron.
- The generalization is not global, so useful interpolation will only occur if there are enough training points—points further apart than the local generalization distance will not be correctly interpolated.
- The input-to-output mapping is discontinuous, without analytical derivatives, although this can be remedied with higher order CMACs [63].
- Selection of CMAC parameters to prevent excessive hash collision can be a large design problem. If there are a limited number of physical weights available then it is difficult to do without knowledge of the input signal coverage, so trial-and-error must usually be used. If physical weights are plentiful then it is acceptable to use far more than are really necessary and not worry about hash collisions.

Despite these problems, the CMAC is extremely useful in real-time adaptive control because of its speed. Far greater processing power would be required when using, say, an MLP network where every weight is involved in every mapping and training operation.

The next chapter will show how the CMAC can be used as the adaptive component in a simple "feedback-error" control system, and the chapter after that will show how the CMAC can be extended to make the FOX controller, which can be used to solve some problems that are beyond the capabilities of feedback-error.

Chapter 4

Feedback-error control

4.1 Introduction

This chapter explains the feedback-error (FBE) control scheme originally described by Kawato [60, 87, 38]. FBE is a widely used neural network based controller which learns to generate the correct control inputs to drive a system's state variables along a desired trajectory. It has been successfully used in a number of control applications, including robot arm control [87], an automatic car braking system [92], and learning nonlinear thruster dynamics to control an underwater robotic vehicle [131].

FBE was originally intended by Kawato to be an abstract model of some of the low level motor areas of the brain, although in the engineering context it has outgrown its biological relevance. The motivation for studying FBE here is that it is a precursor to the FOX controller, with some of the same operational properties (although FOX is far more flexible).

4.2 The basics

This section describes the essentials of FBE control. The arguments are based around a continuous-time system, but identical arguments apply to discrete-time systems. The typical FBE system is shown in figure 4.1. The system being controlled, F, obeys the dynamical equation:

$$\dot{y} \quad = \quad F(y, x) \tag{4.1}$$

In general no restrictions are placed on the function F, but this work is mostly concerned with physical mechanical systems such as robotic manipulators. The vector $y(t)$ is the controlled system's state vari-

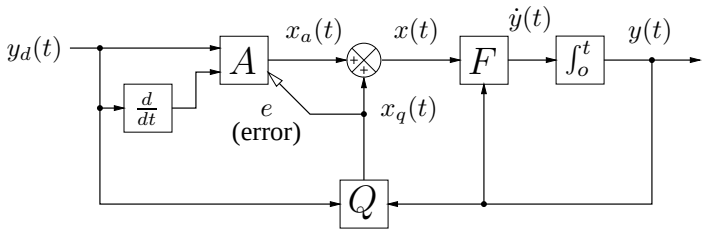


Figure 4.1: *The feedback-error controller.*

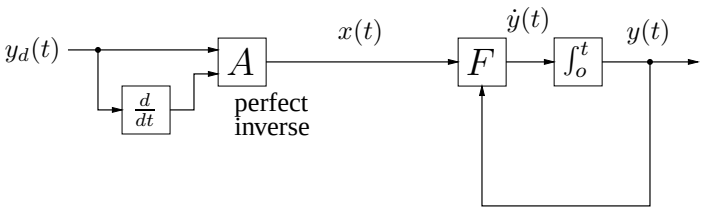


Figure 4.2: *The feed-forward path.*

ables (e.g. positions and velocities) and $x(t)$ is the vector of control variables (e.g. forces or torques). The vector $y_d(t)$ is the *desired* value of $y(t)$, supplied by some external agent. The purpose of the controller, of course, is to make sure $y(t)$ follows $y_d(t)$.

The controller has two main components. First, there is a feed-forward path through A, which is a stateless trainable module (such as a CMAC) that implements

$$x_a = A(y_d, \dot{y}_d) \tag{4.2}$$

Second, there is a feedback path through Q, which is a stateless fixed controller that implements

$$x_q = Q(y_d, y) \tag{4.3}$$

To understand figure 4.1 the feed-forward and feedback paths are examined separately. The convention used is that control operation starts at time $t = 0$.

4.3 The feed-forward path

Figure 4.2 shows the feed-forward path of the controller. In this figure

$$\dot{y} = F\Big(y, A(y_d, \dot{y}_d)\Big) \tag{4.4}$$

Now consider this: if $y(0) = y_d(0)$ then it is possible for A to output a value of x which makes $\dot{y}(0) = \dot{y}_d(0)$, assuming that such a value exists (because A is given the value of y that is also given to F). Extending this argument forward in time, if A is a *perfect* inverse model of the dynamic system it can maintain $y(t) = y_d(t)$ indefinitely for $t > 0$. In other words, it can control the system perfectly by itself. For this to work the *initial* desired and actual states are required to be the same ($y(0) = y_d(0)$) because A has no way of knowing the value of $y(t)$ other than the assumption that $y(t)$ *always* equals $y_d(t)$. If $\dot{y}_d$ is not available then the feed-forward controller will not work, because y_d can't supply information about both the current and desired states of the dynamic system.

A purely feed-forward controller is not practical for two reasons. First, the controller A can never be made perfect, and second, any noise or disturbance in the system will cause y and y_d to diverge.

4.4 The feedback path

Figure 4.3 shows the feedback path of the controller, which implements

$$x_q = Q(y_d, y) \tag{4.5}$$

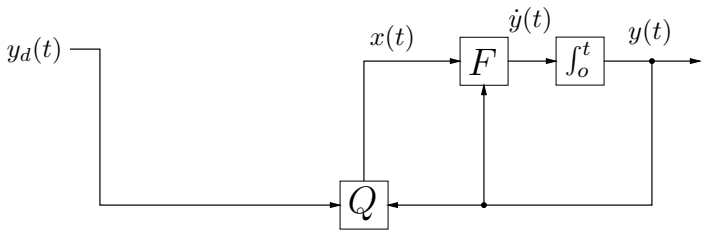


Figure 4.3: *The feedback path.*

The form of Q is arbitrary, although it is usually something like a simple proportional-plus-derivative controller. If Q is chosen well enough then y will tend towards y_d over time. With only the feedback path theoretical perfection can not be achieved (as with the feed-forward path), but practical, stable and robust control is possible.

4.5 Feed-forward and feedback together

To get the best of both types of control the feed-forward and feedback paths can be put together as shown in figure 4.1. The feed-forward path provides instant control signals (x_a) to drive y along the trajectory y_d. The feedback path stabilizes the system by compensating for any imperfections in A or any disturbances. Q provides control signals (x_q) to restore y to the set-point y_d when the two diverge.

If the dynamic system is nonlinear then a linear Q can only be chosen optimally for one operating point. It will be assumed that F is such that a Q can be found that works for the entire state space (albeit with varying degrees of efficiency). Systems which meet this constraint include robotic manipulators. This assumption will be important when FBE training is considered.

4.6 Feedback-error training

In general it is hard to determine the function A analytically for a dynamic system. Even if F is known exactly (an unlikely prospect for all but the simplest systems), computing an inverse has many problems. Thus A is made adaptive (i.e. parameterized by some weight vector w). In practice A is usually some kind of neural network.

The innovation of feedback-error control is that the training (error) signal e given to A is just the feedback signal x_q (hence the name "feedback-error"). Here is the reasoning behind this: if $x_q = 0$ then y will normally equal y_d. In this case the system is perfectly controlled, so the desired error signal is also zero because no adjustment to A is necessary. If $x_q \neq 0$ then the feedback controller is trying to correct the state y to match y_d. For better control, this correction should have been applied by the feed-forward controller some time earlier. Thus x_q is given as an error signal to A so that at this point in the future A will apply a better signal x_a and Q will have less to do (i.e. x_q will be smaller). The signal x_q has the interpretation "this is what x_a should have been". This relies on the generalization properties (local or global) of A. A is trained to minimize x_q and thereby become the inverse of F.

During training there is a transfer of control influence from the "unskilled" feedback controller loop (x_q) to the "skilled" feed-forward controller (x_a). Note that during training, x_a is moved in the *direction* of e (i.e. the error signal is not a target). Now, if

$$x_a = A(y_d, \dot{y}_d, w) \tag{4.6}$$

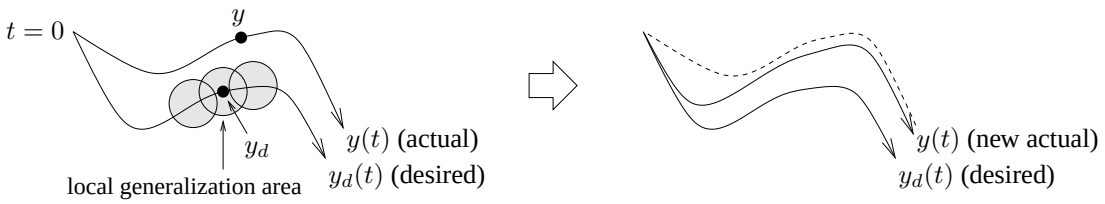


Figure 4.4: *How FBE controller training works.*

where w is a vector of weights controlling A, and if it is assumed that $e = \mathrm{d}E/\mathrm{d}x_a$ (where E is some hypothetical global scalar error) then the gradient descent training algorithm is

$$\dot{w} \quad = \quad \alpha \left[\frac{\mathrm{d}\, x_a}{\mathrm{d}\, w} \right]^T \cdot e \tag{4.7}$$

Here α is a learning rate constant. Note that for the controller to be effective over a certain volume of the state space, the states experienced during training must cover that volume.

4.7 Why it works

Why is the FBE training scheme successful? Figure 4.4 shows how a possible state-space trajectory for y changes during training. Assume that the controller is being repeatedly trained for the same trajectory $y_d(t)$, and that $y(0) = y_d(0)$. Assume also that A implements local generalization so that the training signal $e(t)$ affects a small region around the input point in A (this assumption is not strictly necessary but it will clarify the argument). During each training cycle the same trajectory $(y_d(t), \dot{y}_d(t))$ will be observed by A, so in effect a function of one time input is being trained.

In the early training cycles, y and y_d will start off the same and then gradually diverge (though not too much because of the feedback path). Now, if $y \neq y_d$ near the start of the trajectory, a nonzero error signal x_q will be associated with y_d. When this same desired trajectory is observed in subsequent training runs, the improved x_a will be asserted by A, and the divergence of y from y_d will be reduced. The argument proceeds by induction: as each part of the trajectory $y(t)$ converges to $y_d(t)$, the conditions are set for the convergence of later parts of $y(t)$. Thus the entire system will eventually converge. This argument can be extended to the general case where y_d is free roaming and not going through identical cycles.

4.8 Some other FBE features

4.8.1 Step changes in y_d

Step changes in the desired state y_d will not be handled well with this system, as then $\dot{y}_d$ would be infinite. Instead a trajectory with a smooth first derivative must be computed and passed to the controller.

4.8.2 Unskilled feedback

When the controller is fully trained, feedback has little effect on the control performance. This is acceptable in a "perfect" system, because then y always equals y_d. But in a real system, noise and other disturbances will affect y so that $y \neq y_d$. This is a problem, because the feed-forward controller does

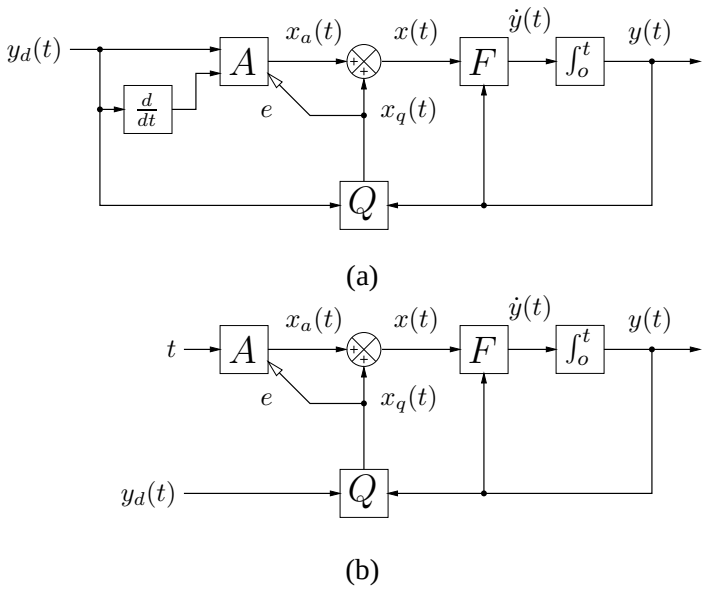


Figure 4.5: *(a) Standard feedback-error (FBE) control. (b) The inputs to A can be replaced by the time t if y_d is always the same one-dimensional trajectory.*

not get any information about the true state of y, it just assumes $y = y_d$. Thus any discrepancies must be dealt with by the "unskilled" feedback controller. So, although the feed-forward controller will cope with the system nonlinearity, noise performance and disturbance rejection are dependent on Q alone.

4.9 An example

It will now be demonstrated how FBE performs on a simple second order system, which corresponds to a unit mass at position p being driven along a straight line by a force x:

$$y = \begin{bmatrix} p \\ \dot{p} \end{bmatrix} \quad , \quad \dot{y} = \begin{bmatrix} \dot{p} \\ \ddot{p} \end{bmatrix} = F(y, x) = \begin{bmatrix} \dot{p} \\ x \end{bmatrix} \tag{4.8}$$

Q is a proportional-plus-derivative feedback controller that tries to get the position p and velocity $\dot{p}$ equal to their reference values:

$$x_q = k_p(p_d - p) + k_v(\dot{p}_d - \dot{p}) \tag{4.9}$$

where k_p and k_v are gain constants. The adaptive controller A is a CMAC neural network.

First a simplification will be made to the standard FBE controller shown in figure 4.5a. If the FBE system is repeatedly trained for the same reference trajectory $y_d(t)$ then A's input $[y_d(t), \dot{y}_d(t)]$ will describe a one-dimensional path parameterized by time t. Thus the inputs to A can be replaced by the time t to get effectively the same system, as shown in figure 4.5b. This simplification will help the following discussion.

Figure 4.6 shows the result of 8000 training iterations with a continuous reference trajectory that has $y_d(0) = y(0)$. The system parameters are given in the figure caption. Note that one iteration is a single run through the entire reference trajectory. The FBE controller is successful in this case—the

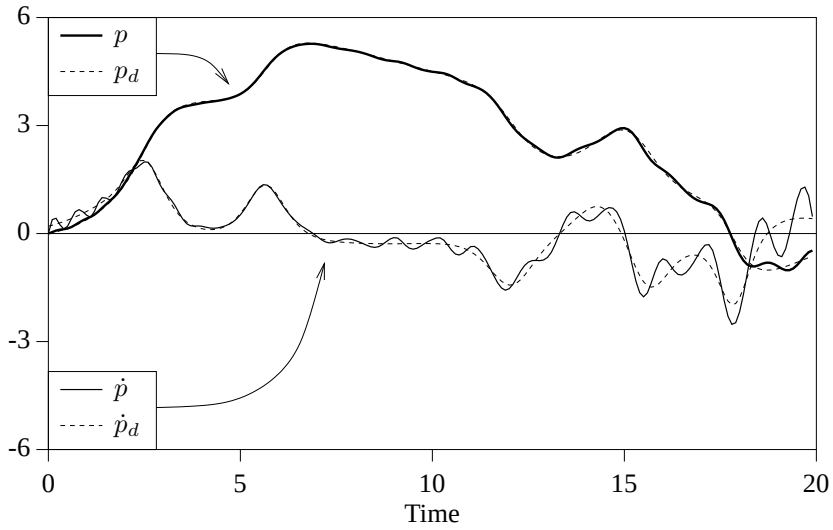


Figure 4.6: *Performance of the FBE second order example system with a continuous reference trajectory that has $y_d(0) = y(0)$. Convergence has almost been achieved after 8000 training iterations. A 200 time step Euler simulation was used with a step size of 0.1. The controller A was a single input CMAC with 200 input quantization levels and $n_a = 10$. The CMAC was trained with a learning rate of 0.05. The feedback controller had $k_p = k_v = 0.5$.*

position p and velocity $\dot{p}$ have mostly converged to their references (although the velocity still has some unconverged oscillations about its reference).

As has already been explained, FBE does not achieve convergence along the entire trajectory at once. Instead, earlier times converge before later times. This is demonstrated in figure 4.7, which shows the same system as in figure 4.6 except the graph was generated after only 600 training iterations. The position and velocity track their reference values for early times, but diverge for later times.

4.10 Arbitrary reference trajectory

FBE convergence requires two things:

1. That $y_d(0) = y(0)$, i.e. that the reference trajectory starts at the actual system starting state. In general, any point where y is constrained to be different from y_d will be incorrectly trained, i.e. there will be divergence instead of convergence.

2. That the reference trajectory is continuous, i.e. that its derivative is bounded above and below. This is obviously necessary if the reference derivative must be computed. However, it is still a requirement in the modified FBE system where t is A's only input, because of the previous requirement. If y_d has a discontinuity then there will be a region after that where y will not be able to match the desired state.

Figures 4.8–4.10 show what happens when these requirements are violated.

Figure 4.8 shows the result of 1000 training iterations when the actual and reference starting states differ. The position oscillates around the reference, and the oscillations increase with further training.

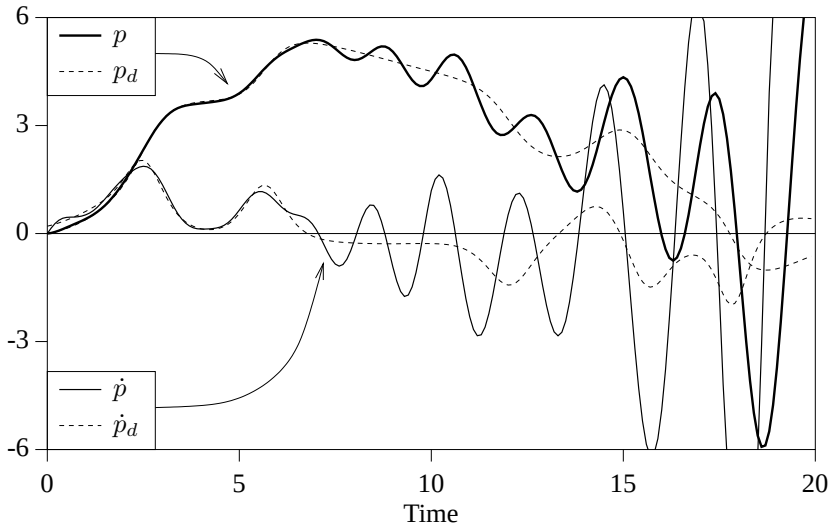


Figure 4.7: *Partial training of the system described in figure 4.6 (600 iterations). Convergence has not been achieved for later times.*

Figure 4.9 shows the result of 1000 training iterations with a discontinuous reference trajectory. The position has mostly converged to its reference but the velocity is oscillating about its reference, especially near the discontinuity. Figure 4.10 shows the same system after 3700 training iterations. The velocity oscillations have increased and are causing noticeable oscillations in position as well. This system continues to diverge with further training.

Why are the reference trajectory requirements necessary to prevent divergence? Consider the points where y is somehow forced to be different from y_d (these will be called "mismatch points"). This obviously happens when the starting reference and actual states differ. It also happens when there is a reference trajectory discontinuity, because no amount of control effort is sufficient to track such a step change. At the mismatch points the error e will always be nonzero. The error e integrates x_a, so x_a at these points will get larger each time the trajectory passes through them. Now, x_a acts to restore the actual state to the reference, but there is no "braking" force applied and so the reference is overshot. The training process does not provide any limiting influences on x_a, so the overshoot gets larger over time. The effects of the overshoot propagate out from the mismatch points, resulting in oscillations. This effect will be called the "over-training problem".

To clarify the problem: When the actual state is close to the reference trajectory, the error value x_q has the meaning "this is what x_a should have been". Elsewhere x_q is just acting as a restoring force. These two different meanings of the error signal (corrective and restorative) conflict.

4.11 Output limiting

One way to solve the over-training problem is to limit the magnitude of x_a somehow. This can be done by training x_a towards zero after each time step (this is called "output limiting"). Thus there will be two competing effects at the mismatch point: the error training from e and the target training towards zero. Eventually an equilibrium will be reached, at which point training will have converged.

This is demonstrated in figure 4.11, which shows the system of figure 4.6 with a reference discon-

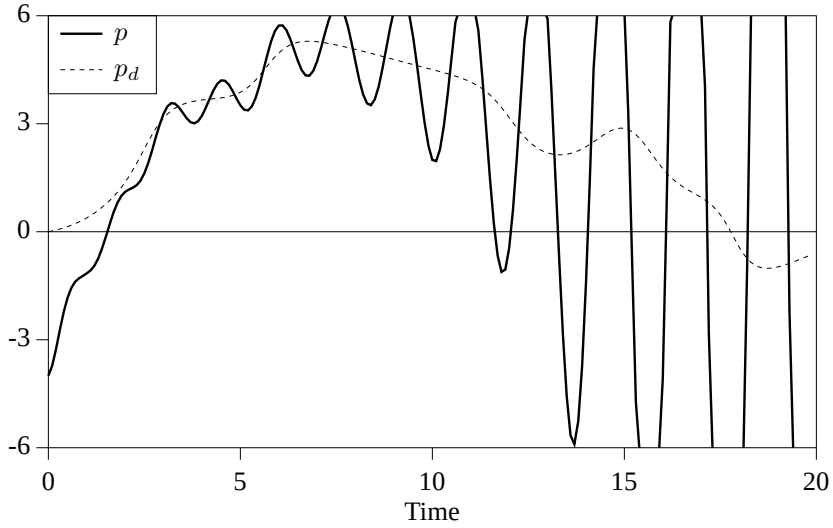


Figure 4.8: *The system of figure 4.6 when the actual and reference starting states differ (1000 iterations).*

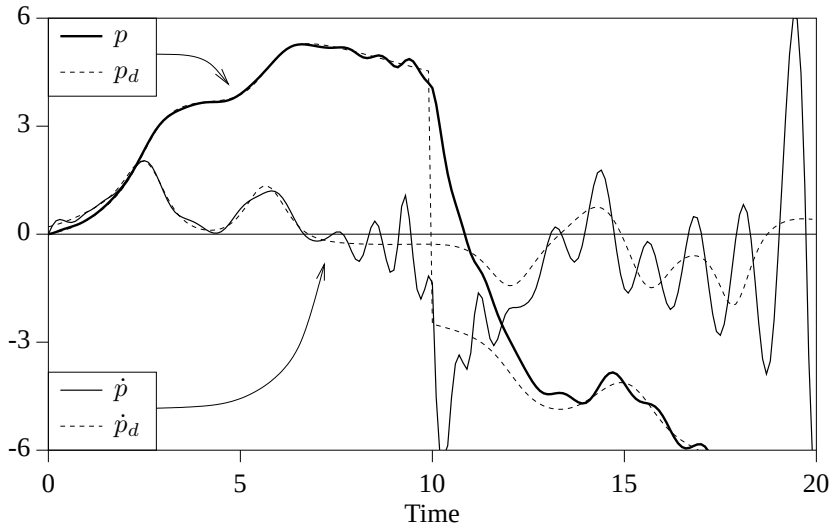


Figure 4.9: *The system of figure 4.6 when the reference trajectory has a discontinuity at* $t = 10$ *(1000 iterations).*

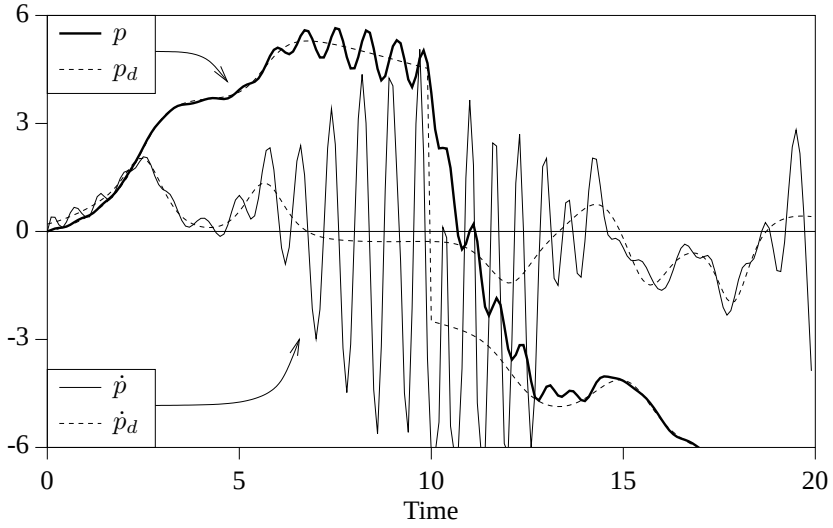


Figure 4.10: *The system of figure 4.6 when the reference trajectory has a discontinuity at* $t = 10$ *(3700 iterations).*

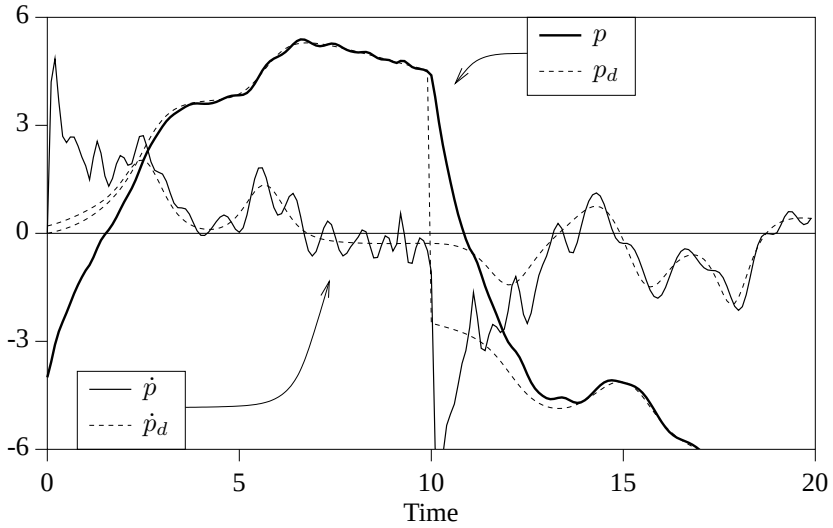


Figure 4.11: *The system of figure 4.6 with a reference discontinuity, an incorrect starting state, and output limiting with a learning rate of 0.002 (5000 iterations).*

tinuity and an incorrect starting state. Output limiting has been used with a learning rate of 0.002. The graph was generated after 5000 iterations, at which point convergence had been achieved. The position tracks its reference reasonably well, except around the mismatch points. The rate at which the position reacquires the reference after a reference discontinuity is dependent on the output limiting learning rate.

4.12 Conclusion

The feedback-error (FBE) control scheme has been described, and it has been shown that its utility is limited by the fact that it requires (1) a known starting state and (2) a continuous reference trajectory to be generated by an external agent. These problems can be partially solved by output limiting, although this is unsatisfactory.

FBE is a starting point for the development of the FOX controller: the next chapter will derive FOX from a first-principles analysis of the problem that FBE is trying to solve. It will be shown that the FOX algorithm can be regarded as a generalization of FBE. Experiments on FOX will subsequently show that FBE has some hidden problems: in particular, it is only capable of controlling systems that have a "zero-order eligibility model".

Chapter 5

Theory of the FOX controller

5.1 Introduction

This chapter describes a CMAC-based adaptive controller called FOX[1]. FOX will learn to generate control signals that minimize an "error" function of the system's behavior. FOX uses a gradient descent approach to perform the optimization. As shown in Appendix B, some standard gradient descent training methods on adaptive dynamic systems can require a large amount of time or memory space. In contrast, the FOX algorithm is not much slower than the basic CMAC algorithm (per controller time step).

FOX implements eligibility-based reinforcement learning [10] because it maintains auxiliary values for each CMAC weight which measure the eligibility of that weight for modification. Previous CMAC-based reinforcement learning techniques [67] have been slow, and there is little guidance available for choosing their eligibility update equations [68]. In contrast, the FOX algorithm's eligibility update equations are formally derived in section 5.4.

The description of FOX below is motivated by considering the feedback-error (FBE) controller. The standard FBE control scheme described in Chapter 4 requires a reference trajectory which must be both continuous and coincident with the system's starting state. It has already been seen that if these requirements are not met then over-training results and the system becomes unstable. It would be good to overcome this problem for two reasons. First, being able to guarantee stability in a wider range of cases implies more robust control. Second, the reference trajectory limitations mean that FBE only solves half of the intelligent control problem. To explain this, if the system's starting state is arbitrary (or unknown), then a continuous reference trajectory must be generated to get the system from where it is to the desired state. Generating such a trajectory analytically, say as the path that optimizes some error criterion, is the other half of the problem. Ideally the designer would like to specify a single reference trajectory and then have the system converge to it in some well defined manner. Reference trajectory discontinuities should also be handled this way.

One solution to this problem, output limiting, has already been explored. This chapter will first look at another solution, called "weight eligibility". A hand-waving analysis of this scheme will demonstrate its advantages, then it will be shown how it can be implemented in the CMAC. Then the FOX algorithm will be formally derived by considering how to optimize an error function of the controlled system's inputs and outputs. This derivation is not based on FBE, but it will be shown that the resulting FOX system has many similarities to the FBE-plus-eligibility controller. It will also be shown that straight

[1] What does 'FOX' mean? Well, I thought it was a **F**airly **O**bvious e**X**tension to the CMAC.

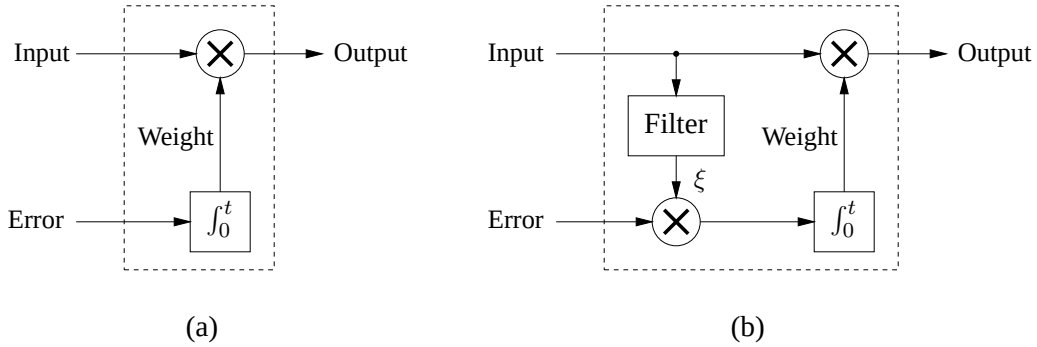


Figure 5.1: *(a) A normal neural network "synapse". (b) A modified synapse which has weight eligibility ξ.*

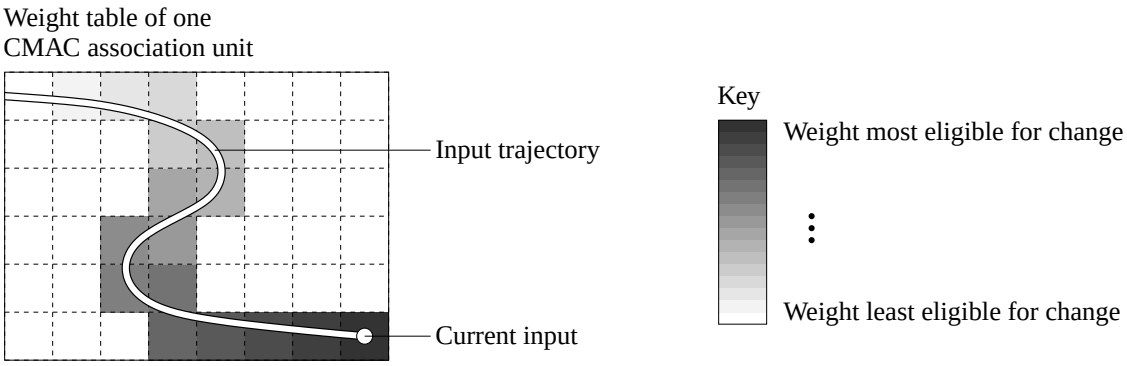


Figure 5.2: *How eligibility is implemented in a two-input binary CMAC.*

FBE is unable to control many kinds of systems. An efficient implementation for the FOX algorithm will be described which avoids having to update every weight's eligibilities after each CMAC mapping.

The derivation of the FOX algorithm is based in part on the author's previous work: [113] for the unfolded linear approximations, and [115] for some of the CMAC-with-eligibility concepts.

5.2 FBE and weight eligibility

Eligibility is an idea that has been used for many years as part of the reinforcement learning paradigm [72]. The basic idea is that each weight in a neural network is given an associated value called its "eligibility". When the weight is used (i.e. when its value makes a contribution to the network output) the eligibility is increased. Thereafter it decays toward zero. Figure 5.1a shows a normal neural network "synapse" in which a weight multiplies some input to get some output. The weight is usually the integrated value of some error signal, which is somehow derived from the global error signals given to the network. Figure 5.1b shows how the synapse is modified for eligibility. The input is filtered to obtain an eligibility signal ξ, which is multiplied by the error and the result integrates the weight. Many different forms of the eligibility filter have been proposed to solve specific problems — see [72] for a review — but it is usually a low pass filter of some description.

Figure 5.2 shows how eligibility is implemented in a two-input binary CMAC. In a binary CMAC all

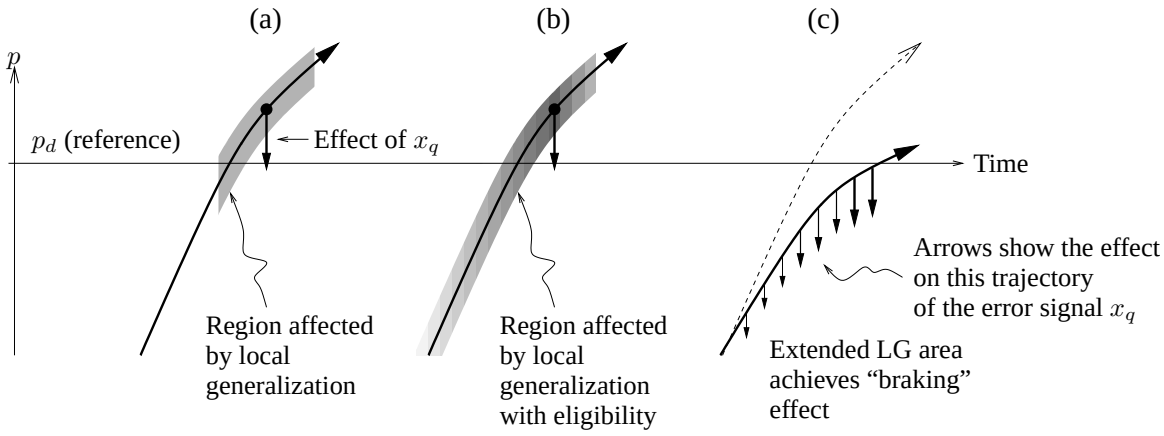


Figure 5.3: *For the example FBE system, how adding eligibility to the standard CMAC local generalization increases the region over which x_q acts, achieving a "braking" effect which helps to prevent instability.*

the synapse inputs will be either zero or one. All weights along the input trajectory will have some degree of eligibility. Training should change all eligible weights by an amount proportional to their eligibility, i.e. the error signal is only effective at integrating the weight while the eligibility is nonzero (while the input is one and for some time thereafter).

Consider the simple second order FBE example system presented in the previous chapter. Eligibility can improve the system in two ways. First, it can increase the system stability (reduce the oscillations) by increasing the region over which x_q acts, achieving a "braking" effect. Figure 5.3a shows the system state approaching and overshooting the reference. At any point the feedback-error signal x_q influences a small area of the trajectory corresponding to the CMAC local generalization (LG) region. Figure 5.3b shows that this area expands when eligibility is used, because weights influencing the older trajectory have remained eligible for modification by the error signal. The result is that on the next training iteration the braking force arising from x_q is able to be applied earlier, which is more appropriate for reducing the overshoot.

The second improvement is that cause and effect are better associated in the system. This is demonstrated in figure 5.4. Curve 1 shows a position reference with a step change. Curve 2 is a typical FBE response (when output limiting is used). Local generalization allows the system state to slightly anticipate the reference change. Curve 3 shows what can happen if eligibility is used: the system state may be able to greatly anticipate the change, because the training induced by the error signal affects the system further back in time. This helps to reduce the error between the actual and reference trajectories.

5.3 Introduction to FOX

The basic discrete-time FOX system shown in figure 5.5 will now be considered. This consists of a process F being driven by a CMAC controller. The block F is the controlled system (or "plant") that implements

$$\mathbf{y}_{i+1} \quad = \quad F(\mathbf{y}_i, \mathbf{x}_i) \tag{5.1}$$

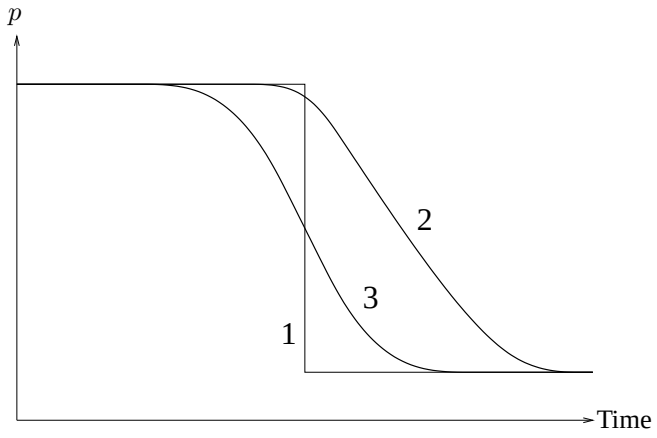


Figure 5.4: *For the example FBE system, how eligibility can give better prediction.*

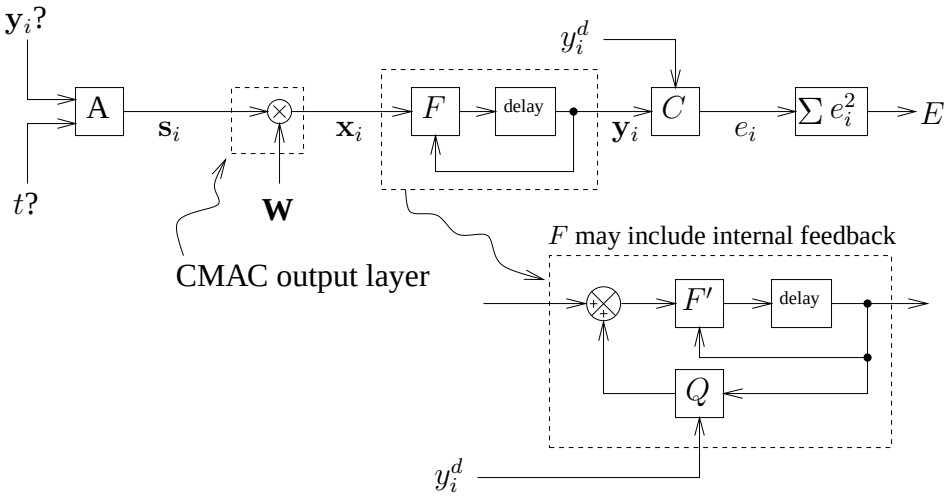


Figure 5.5: *The FOX controller and the controlled system.*

Symbol	Type	Size	Description
$F(\mathbf{y}_i, \mathbf{x}_i)$	f	–	The system function, which takes the current state and control input and generates the next time step's state.
$C(\mathbf{y}_i)$	f	–	The "critic" function, which generates a scalar error value at each time step.
$\mathbf{y}_i$	v	$n_y \times 1$	The system state vector for time i.
$\mathbf{x}_i$	v	$n_x \times 1$	The system control input vector for time i (the output of the CMAC controller).
$\mathbf{s}_i$	v	$n_s \times 1$	A vector of CMAC association-unit "sensors" that encodes the current system state.
W	m	$n_x \times n_s$	The matrix of CMAC weights. Note that $\mathbf{x}_i = W\,\mathbf{s}_i$.
e_i	s	1×1	A scalar error value generated by the critic function for time i.
y_i^d	s	1×1	A position reference (for time i) used in the calculation of e.
E	s	1×1	The total error of the entire system.
w_{pq}	s	1×1	Element (p,q) of the matrix W.
ξ_i^{pq}	v	$n_y \times 1$	The eligibility value for weight w_{pq}.
$\hat{\mathbf{s}}_i^{pq}$	v	$n_x \times 1$	A synonym for $\mathrm{d}\mathbf{x}_i/\mathrm{d}w_{pq}$. $\hat{\mathbf{s}}_i^{pq}$ is all zero except for its p'th element which is equal to the q'th element of $\mathbf{s}_i$.
n_w	s	1×1	Total number of weights, $n_w = n_s\,n_x$.
n_a	s	1×1	Number of association units in the CMAC, i.e. the number of nonzero elements in $\mathbf{s}_i$.
A	m	$n_y \times n_y$	A system matrix for the linear system F^*.
B	m	$n_y \times n_x$	A system matrix for the linear system F^*.
C	m	$1 \times n_y$	The matrix for the simple linear critic function.
α	s	1×1	The main FOX learning rate.

Table 5.1: *Definitions of some symbols in figure 5.6 and elsewhere in this chapter. This is a subset of the symbols defined in the nomenclature section of this thesis. The types are function (*f*), vector (*v*), matrix (*m*) and scalar (*s*).*

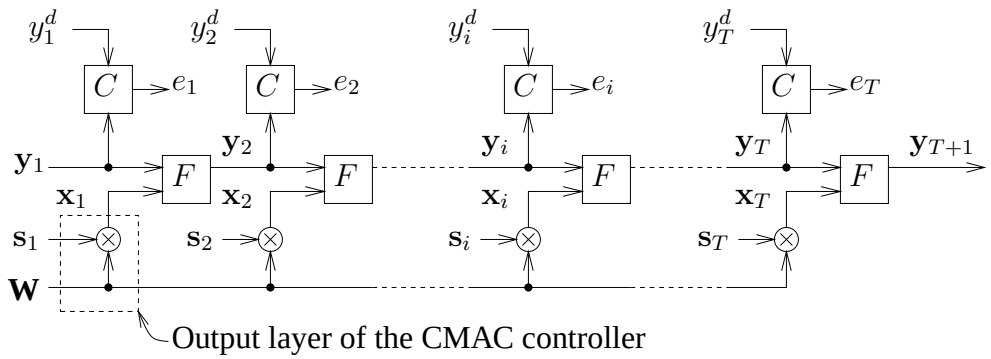


Figure 5.6: *Figure 5.5 unfolded over time.*

As shown in the figure the block F can also incorporate any additional feedback controller Q that the basic system might have (this is equivalent to the feedback controller in the FBE control scheme). The CMAC has been split into two parts in the figure:

1. The block A which represents the input layers (or association unit transformations). At each time step this block reads the system state $\mathbf{y}$ (or perhaps the current time t) and encodes it in the "sensor" vector $\mathbf{s}$.

2. The output layer which multiplies the vector $\mathbf{s}$ with the weight matrix W to get the output $\mathbf{x}$ ($\mathbf{x} = W\,\mathbf{s}$).

At each time step the "critic" block C computes a scalar error value e_i that depends on the current state. The squares of these error values are summed over time to get the global error value E. The goal of the FOX algorithm is to adjust the weights W such that the error E is minimized—in other words, this is an optimal control problem.

Figure 5.6 shows the same system unfolded over time steps $1 \ldots T$ (the symbols used are defined in table 5.1, also see the nomenclature section of this thesis). This is the figure that will be referred to in the following derivations.

Note that the CMAC controller has no internal state variables. Controller state can give extra flexibility, but for the moment it will be assumed that any extra dynamics are incorporated into the F block.

5.3.1 The critic function

The total error E is given by

$$E \;=\; \sum_{i=1}^{T} e_i^2 \tag{5.2}$$

The critic function is chosen so that when E is minimized the system achieves some desired behavior. It will be assumed henceforth that the desired behavior is to track a scalar reference position y^d, thus:

$$e_i \;=\; C\,\mathbf{y}_i - y_i^d \tag{5.3}$$

Here C is a $1 \times n_y$ matrix that selects whatever element of y corresponds to a position. This style of error function has several limitations. It can not be used if more than one component of $\mathbf{y}$ is required to follow a reference. It also does not limit the control force $\mathbf{x}$. These limitations will be relaxed later when the fundamental FOX algorithm has been developed.

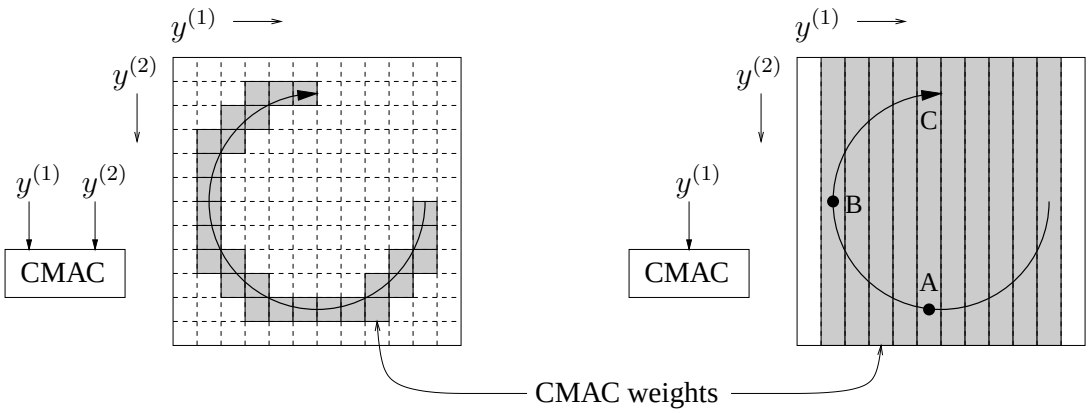


Figure 5.7: *How the sensor vectors can become correlated if the CMAC input does not include enough information.*

5.3.2 The sensor vector and CMAC inputs

For the binary CMAC the sensor vector components are zero or one. In one sense, the ideal set of sensor vectors $\mathbf{s}_i$ would be such that each one selected different columns of the matrix W. This would mean that a different set of weights controls each $\mathbf{x}_i$, so the training process has the greatest amount of flexibility when converging to an optimal control strategy, i.e. independent outputs can be achieved for each time step.

In practice the sensor vectors will usually have overlapping components from one time step to the next, so the training flexibility is reduced, i.e. the sensors $\mathbf{s}$ will not perfectly distinguish the current state from the others. This effect can be an advantage, because the CMAC output will usually be smoother (less varying), thus the CMAC does not learn unjustifiably complex control outputs, which can improve the system's controllability.

In terms of training it makes little difference whether the CMAC input is derived from the current state $\mathbf{y}$, the desired state $\mathbf{y}^d$, or the time t. All that is required is that, for the expected system trajectories, the input is sufficiently variable that the sensor vectors are not overly correlated from one time step to the next. Thus the system designer may select the CMAC input based on other considerations. If the current state $\mathbf{y}$ is used the FOX system is said to be operating in *feedback mode*, and if $\mathbf{y}^d$ or t are used it is said to be operating in *feed-forward mode*.

Figure 5.7 shows what *not* to do. The diagram on the left shows an AU table of a two-input CMAC. At each part of the example trajectory different weights are selected, so that the CMAC output is sufficiently flexible. The diagram on the right shows the AU table when only one of the inputs is used. The two parts of the trajectory between A–B and B–C use the same weight values, so training will not be able to generate independent outputs for them.

5.4 Derivation of the training algorithm

Now the FOX training algorithm will be derived. The purpose of the algorithm is to modify W using gradient descent so that the error E is minimized[2]. This section has a lot of equations, with little

[2]It is instructive to compare this derivation to the dynamic programming solution to the optimal control problem (see Appendix D).

explanation—the next section will say what it all means. The gradient descent process for this system is:

$$w_{pq} \leftarrow w_{pq} - \alpha \frac{\mathrm{d}\, E}{\mathrm{d}\, w_{pq}} \tag{5.4}$$

where w_{pq} is an element of the matrix W and α is a scalar learning rate. This equation gives the modification made to the weight w_{pq} as a result of one iteration of learning. Now, from the chain rule:

$$\frac{\mathrm{d}\, E}{\mathrm{d}\, w_{pq}} = \sum_{i=1}^{T-1} \frac{\mathrm{d}\, E}{\mathrm{d}\, \mathbf{x}_i} \cdot \frac{\mathrm{d}\, \mathbf{x}_i}{\mathrm{d}\, w_{pq}} \tag{5.5}$$

$$= \sum_{i=1}^{T-1} \left[2 \sum_{k=i+1}^{T} e_k \frac{\mathrm{d}\, e_k}{\mathrm{d}\, \mathbf{x}_i} \right] \frac{\mathrm{d}\, \mathbf{x}_i}{\mathrm{d}\, w_{pq}} \tag{5.6}$$

Note that $(T-1)$ is the limit on the outer sum, not T, because $\mathbf{x}_T$ has no effect on e_T. $\mathrm{d}e_k/\mathrm{d}\mathbf{x}_i$ can be calculated using forward propagation, i.e.,

$$\frac{\mathrm{d}\, e_{i+1}}{\mathrm{d}\, \mathbf{x}_i} = \frac{\partial\, e_{i+1}}{\partial\, \mathbf{y}_{i+1}} \cdot \frac{\partial\, \mathbf{y}_{i+1}}{\partial\, \mathbf{x}_i} \tag{5.7}$$

and

$$\frac{\mathrm{d}\, e_{i+2}}{\mathrm{d}\, \mathbf{x}_i} = \frac{\partial\, e_{i+2}}{\partial\, \mathbf{y}_{i+2}} \cdot \frac{\partial\, \mathbf{y}_{i+2}}{\partial\, \mathbf{y}_{i+1}} \cdot \frac{\partial\, \mathbf{y}_{i+1}}{\partial\, \mathbf{x}_i} \tag{5.8}$$

so

$$\frac{\mathrm{d}\, e_{i+k}}{\mathrm{d}\, \mathbf{x}_i} = \frac{\partial\, e_{i+k}}{\partial\, \mathbf{y}_{i+k}} \left[\frac{\partial\, \mathbf{y}_{i+k}}{\partial\, \mathbf{y}_{i+k-1}} \cdots \frac{\partial\, \mathbf{y}_{i+2}}{\partial\, \mathbf{y}_{i+1}} \right] \cdot \frac{\partial\, \mathbf{y}_{i+1}}{\partial\, \mathbf{x}_i} \qquad (k > 0) \tag{5.9}$$

Equation 5.6 is transformed into a form suitable for online training (i.e., incremental accumulation of the outer sum) by reversing the order of summation (swapping the variables i and k):

$$\frac{\mathrm{d}\, E}{\mathrm{d}\, w_{pq}} = 2 \sum_{k=2}^{T} e_k \left[\sum_{i=1}^{k-1} \frac{\mathrm{d}\, e_k}{\mathrm{d}\, \mathbf{x}_i} \frac{\mathrm{d}\, \mathbf{x}_i}{\mathrm{d}\, w_{pq}} \right] \tag{5.10}$$

Now for a key step—to determine the meaning of equation 5.10, and to derive a practical algorithm, F is *approximated* by a linear system F^*:

$$\mathbf{y}_{i+1} = A\, \mathbf{y}_i + B\, \mathbf{x}_i \tag{5.11}$$

$$e_i = C\, \mathbf{y}_i - y_i^d \tag{5.12}$$

Of course if F is already linear then this can be an exact model. Combining this with equation 5.9:

$$\frac{\mathrm{d}\, e_{i+k}}{\mathrm{d}\, \mathbf{x}_i} = C\, A^{k-1}\, B \qquad (k > 0) \tag{5.13}$$

thus

$$\frac{\mathrm{d}\, E}{\mathrm{d}\, w_{pq}} = 2 \sum_{k=2}^{T} e_k\, C \left[\sum_{i=1}^{k-1} A^{k-i-1}\, B\, \hat{\mathbf{s}}_i^{pq} \right] \tag{5.14}$$

$$= 2 \sum_{k=2}^{T} e_k\, C\, \xi_k^{pq} \tag{5.15}$$

If $\mathbf{y}_1 = 0$ and the values of $\mathbf{x}_1 \ldots \mathbf{x}_{k-1}$ are known, and $\mathbf{y}_{i+1} = A\,\mathbf{y}_i + B\,\mathbf{x}_i$, then $\mathbf{y}_k$ can be calculated as follows:

$$\begin{aligned}
\mathbf{y}_1 &\triangleq 0 \\
\mathbf{y}_2 &= B\,\mathbf{x}_1 \\
\mathbf{y}_3 &= AB\,\mathbf{x}_1 + B\,\mathbf{x}_2 \\
\mathbf{y}_4 &= A^2B\,\mathbf{x}_1 + AB\,\mathbf{x}_2 + B\,\mathbf{x}_3 \\
&\vdots \\
\mathbf{y}_k &= A^{k-2}B\,\mathbf{x}_1 + A^{k-3}B\,\mathbf{x}_2 + \cdots + AB\,\mathbf{x}_{k-2} + B\,\mathbf{x}_{k-1} \\
&= \sum_{i=1}^{k-1} A^{k-i-1}B\,\mathbf{x}_i
\end{aligned}$$

If $\mathbf{x}_i$ is replaced with $\hat{\mathbf{s}}_i^{pq}$ then $\mathbf{y}_k$ is equal to ξ_k^{pq}.

Table 5.2: *Proof that ξ_k^{pq} is subject to the same dynamics as F^*.*

where

$$\xi_k^{pq} = \sum_{i=1}^{k-1} A^{k-i-1}\,B\,\hat{\mathbf{s}}_i^{pq} \qquad \text{and} \qquad \hat{\mathbf{s}}_i^{pq} = \frac{\partial\,\mathbf{x}_i}{\partial\,w_{pq}} \tag{5.16}$$

$\hat{\mathbf{s}}_i^{pq}$ is all zero except for its p'th element which is equal to the q'th element of $\mathbf{s}_i$. ξ_k^{pq} is the *eligibility* signal. It can be shown that ξ_k^{pq} is a convolution of $\hat{\mathbf{s}}_i^{pq}$ with the impulse response of the system F^*, up to time k (see table 5.2).

In other words, ξ_k^{pq} is subject to the same dynamics as the system's linear approximation, so

$$\xi_1^{pq} = 0 \tag{5.17}$$
$$\xi_{i+1}^{pq} = A\,\xi_i^{pq} + B\,\hat{\mathbf{s}}_i^{pq} \tag{5.18}$$

5.5 Discussion

5.5.1 Algorithm summary

Here is the FOX training algorithm, based on equations 5.4, 5.15 and 5.18:

$$\xi_1^{pq} = 0 \tag{5.19}$$
$$\xi_{i+1}^{pq} = A\,\xi_i^{pq} + B\,\hat{\mathbf{s}}_i^{pq} \tag{5.20}$$
$$w_{pq} \leftarrow w_{pq} - \alpha \sum_{i=2}^{T} e_i\,C\,\xi_i^{pq} \tag{5.21}$$

(note that a factor of 2 has been combined into α). Every CMAC weight w_{pq} requires an associated eligibility vector ξ^{pq}. The *order* of the eligibility model is the size of the matrix A. A zero-order model is a FOX system without any eligibility machinery at all, which is just a straight CMAC (the corresponding zero-order impulse response is just the Dirac delta function $k\delta(t)$, where k is an arbitrary scale factor).

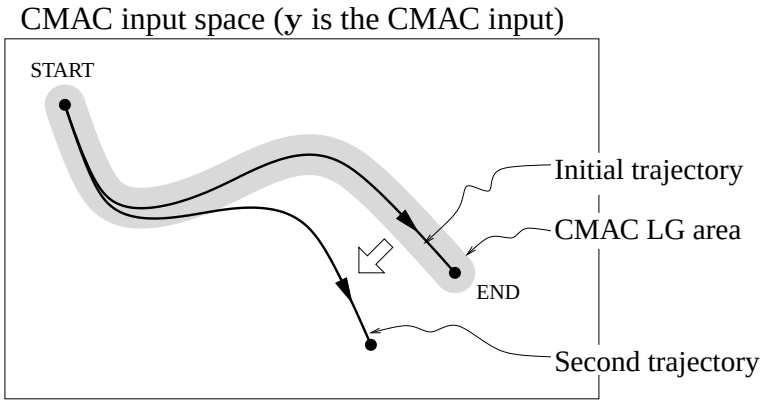


Figure 5.8: *How the* $\mathrm{d}\mathbf{x}/\mathrm{d}\mathbf{y} = 0$ *assumption can be violated in one training iteration.*

Notice that there is a relationship between the two constants α and C : if the magnitude of C is adjusted then α can be changed to compensate. Because of this the convention will be adopted that the magnitude of C is always set to one ($|C| = 1$) and then the resulting α is the main FOX learning rate.

5.5.2 Approximation: $\partial\mathbf{x}/\partial\mathbf{y} = 0$

The derivation of this algorithm has deliberately ignored the influence of the system state $\mathbf{y}$ on the CMAC output $\mathbf{x}$, in other words it has been assumed that $\partial\mathbf{x}/\partial\mathbf{y} = 0$. There are several reasons for this approximation. First, ignoring it makes the FOX algorithm simple. Second, the CMAC input is not necessarily tied to the system state, for example it could be the system time or some other sensor values. Third, the CMAC input to output mapping is discontinuous, so the value of $\partial\mathbf{x}/\partial\mathbf{y}$ is also discontinuous which makes it difficult to use in a training algorithm.

Despite this, the approximation is acceptable. Consider figure 5.8 which shows how a CMAC input-space trajectory is modified over one training iteration (assuming that $\mathbf{y}$ is the CMAC input). Training over the initial trajectory modifies the CMAC weights, so that the second trajectory is different because the CMAC outputs $\mathbf{x}$ are different (assuming the starting point is the same). The fact that a new region of the input space is being traversed will change the output $\mathbf{x}$, quite apart from the fact that the $\mathbf{x}$ values in the local generalization (LG) area of the first trajectory have been changed. This is the effect that is ignored by assuming $\partial\mathbf{x}/\partial\mathbf{y} = 0$.

The degree to which the original and modified trajectories differ depends on the learning rate α. If α is small enough that the modified trajectory is still mostly contained within the LG area of the initial trajectory, then the approximation error will be small enough not to matter. Note that the larger the learning rate, the more likely it is that the gradient descent step will actually increase the global error.

5.5.3 Approximation: instantaneous weight update

Equation 5.21 implies that a sum from times $2 \ldots T$ must be computed before the weights can be updated. This would require an auxiliary sum-accumulation variable for each weight. The following approximation will be made to remove the need for these extra variables: at each time step as each term of the sum is computed it will be added directly to the weight. This approximation is very common in neural network training algorithms, including the standard backpropagation algorithm. If α is small enough then this approximation will have a negligible effect on the result.

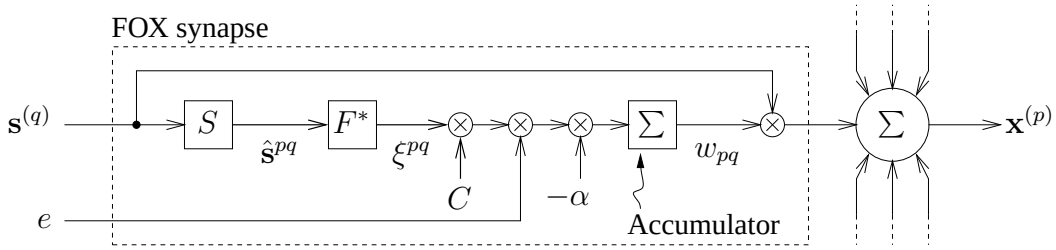


Figure 5.9: *A single "synapse" of the FOX controller, for weight w_{pq}. $\mathbf{s}^{(q)}$ is the q'th element of $\mathbf{s}$, $\mathbf{x}^{(p)}$ is the p'th element of $\mathbf{x}$.*

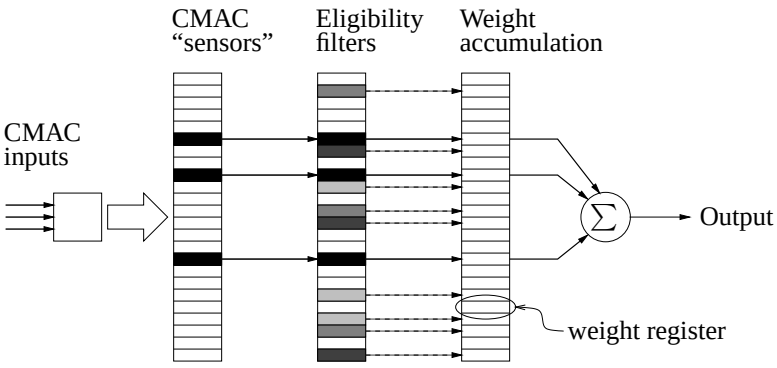


Figure 5.10: *How training is performed, conceptually.*

5.5.4 The synapse mechanism

A schematic of the process used to update a single weight is shown in figure 5.9. In this figure the block S converts the sensor vector element $\mathbf{s}^{(q)}$ into the vector $\hat{\mathbf{s}}^{pq}$ by padding it with zeros. The outputs of the active synapses (those with nonzero $\mathbf{s}^{(q)}$) are summed to produce the CMAC outputs. Compare this with figure 5.1b—it can be seen that the FOX synapse fits the traditional picture of a synapse with eligibility. Figure 5.10 shows how these synapse units are collected together in the CMAC.

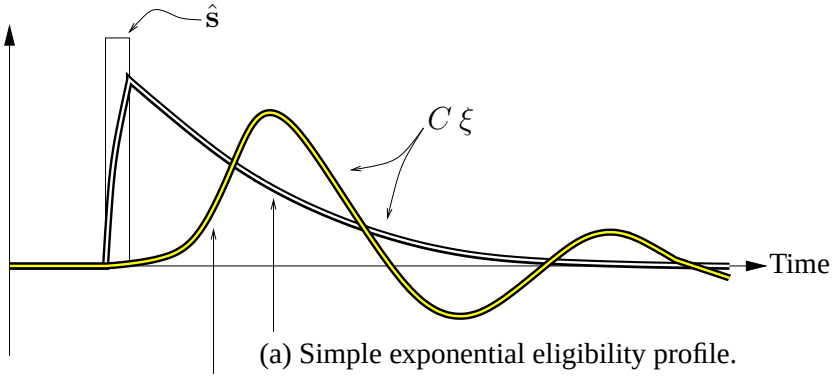
5.5.5 Eligibility profile

The eligibility ξ is a vector, but the important quantity in determining the eligibility's effect on the system is the *scalar* value $C\xi$. The response of $C\xi$ to an impulse in the synapse input $\mathbf{s}^{(q)}$ is called the *eligibility profile*. It is a scalar function of time step i:

$$\text{eligibility profile } (i) = C\, A^i\, B \tag{5.22}$$

There are n_x distinct eligibility profiles in the system, one for each component of $\mathbf{x}$. The i'th eligibility profile is the error that is measured (for a zero reference) when the system F^* is excited with an impulse on input $\mathbf{x}^{(i)}$. The eligibility profiles are defined by the matrices A,B and C. They encapsulate all the information about the system dynamics that is required for training.

Many previous eligibility implementations in reinforcement learning algorithms [72] have used an arbitrarily chosen exponential-decay scalar eligibility profile (figure 5.11a). In contrast the FOX eligibility profiles are selected to match the error-impulse-response of the controlled system, and can be a



(a) Simple exponential eligibility profile.

(b) Complex vector eligibility profile.

Figure 5.11: *Some eligibility profiles.*

much more complicated function (figure 5.11b). Thus FOX allows "higher order eligibility" or "vector eligibility".

Any alternative eligibility filter (F^*,C) that generates a similar profile will work just as well. This fact can be used to simplify F^* for complex systems, and is the basis for design techniques that will be presented later.

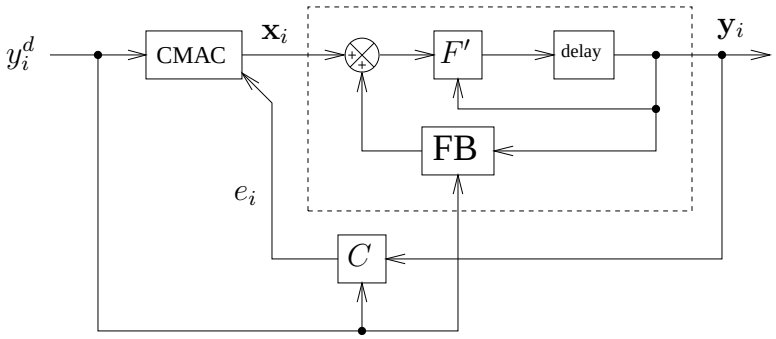
5.6 FOX and FBE

The standard feedback-error controller described in the previous chapter is very similar to a zero-order FOX system. This is demonstrated in figure 5.12 which shows a comparison between the two systems. There are a number of differences, but they are not significant:

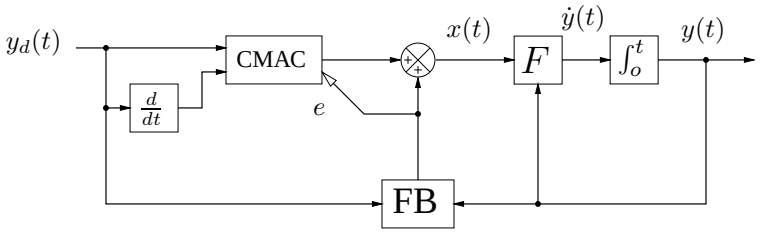
- The FOX system operates in discrete time and the FBE system operates in continuous time. But either system can be easily converted in to the other form.
- The FBE system uses the feedback signal as the error, while the FOX system generates it independently in the function C.
- The FBE CMAC inputs include $\mathrm{d}y_d(t)/\mathrm{d}t$, while the FOX CMAC input only has $\mathbf{y}_i^d$. This allows the FBE system to be trained to follow a $y_d(t)$ trajectory that is different in each iteration while the FOX system's desired trajectory must be the same in each iteration. But either input set can be used for either system.

Thus there is no real difference between the two systems in terms of the effect that training has, so FBE can be considered to be equivalent to a zero-order FOX. This implies that FBE can only provide successful control when the system F plus its feedback controller have a zero-order impulse response $k\delta(t)$ as described above (this means that the effect of the control effort is immediately observable in the sensors).

This is generally true when FBE is being used to generate forces to control the *accelerations* of a mechanical system (as the effects of applied forces can be immediately observed in the system accelerations). Examples of FBE systems where this is true are robot arm control [87], an automatic car braking system [92], and learning nonlinear thruster dynamics to control an underwater robotic vehicle [131].



(a) A zero-order FOX system



(b) A standard feedback-error system

Figure 5.12: *Comparison of (a) a zero-order FOX system and (b) a standard feedback-error system. 'FB' is the "internal" feedback controller for (a) and the feedback controller for (b).*

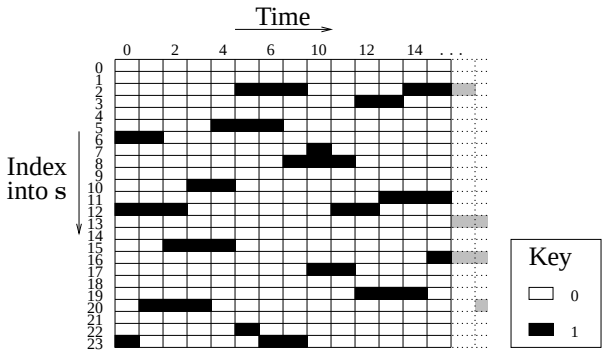


Figure 5.13: *Time evolution example for the vector* $\mathbf{s}$, $n_s = 24$ *and* $n_a = 3$.

It will be shown in the next chapter that if FBE is used to generate forces to control other parameters of a mechanical system (positions or velocities) then over-training and divergence will usually occur, manifested as system oscillations).

5.7 Implementation

A naïve implementation of the training equations is very simple—just update the eligibility state for every weight during each time step. Consider a CMAC with n_w weights ($n_w = n_x n_s$) and n_a association units. To compute the CMAC's output without training (in the conventional way) requires one set of computations per association unit, so the computation required is $\mathcal{O}(n_a)$ per time step. But if eligibilities must be updated as well then one set of computations per *weight* is needed, so the time rises by $\mathcal{O}(n_w)$. A typical CMAC has $n_w \gg n_a$ (e.g. $n_a = 10$ and $n_w = 10,000$), so the naïve approach usually requires too much computation to be practical in an online controller.

The FOX algorithm performs the same computations using a far more efficient approach which eliminates the recalculation of redundant information (in this respect it is like the Fast Fourier Transform algorithm). FOX performs just $\mathcal{O}(n_a)$ extra operations per time step, so the total computation time is still $\mathcal{O}(n_a)$—the same order as the basic CMAC algorithm.

5.7.1 Assumptions

The algorithm described below requires the system F^* to have an impulse response that eventually decays to zero. This is equivalent to requiring that the eigenvalues of A all have a magnitude less than one. This will be called the "decay-to-zero" assumption. The reasons for this assumption will be made clear later. The next chapter will explain how to get around this requirement in a practical system.

5.7.2 Operation

Figure 5.13 shows an example of how the vector $\mathbf{s}$ (the CMAC sensors) changes over time. At any time only n_a (in this case three) vector elements are nonzero. These elements correspond to the activated weights. The activated weight indexes change unpredictably as the CMAC input evolves over time. However, each element of $\mathbf{s}$ spends most of its time at zero, because $n_a \ll n_w$.

Figure 5.14 shows conceptually how an example eligibility filter responds to a typical sensor signal. When the sensor signal is one the eligibility rises, and when it is zero (which is most of the time) the

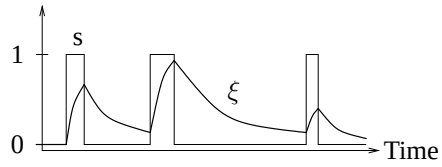


Figure 5.14: *The output (ξ) of a sample eligibility filter being driven by a sensor signal (s).*

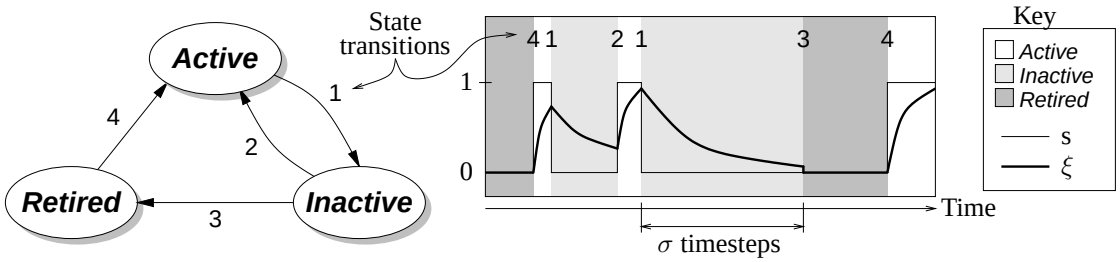


Figure 5.15: *The three states of a FOX weight, and the transitions between them.*

eligibility decays towards zero as required by the decay-to-zero assumption. This behavior inspires division of the weights into the following three categories:

- *Active weights*: where the weight is one of the n_a currently being accessed by the CMAC. There are always n_a active weights.
- *Inactive weights*: where the weight was previously active and its eligibility has yet to decay to zero.
- *Retired weights*: where the weight's eligibility has decayed sufficiently close to zero, so no further weight change will be allowed to take place until this weight becomes active again.

Figure 5.15 shows how a weight makes the transition between these different states. FOX does not have to process the retired weights because their values do not change (their eligibilities are zero and will remain that way) and they do not affect the CMAC output. An active weight turns in to an inactive weight when the weight is no longer being accessed by the CMAC (transition 1 in figure 5.15). An inactive weight turns in to a retired weight after σ time steps have gone past (transition 3 in figure 5.15). The value of σ is chosen so that after σ time steps a decaying eligibility value is small enough to be set to zero. Ways to choose σ will be discussed later. At each new time step a new set of weights are made active. Some of these would have been active on the previous time step, others are transferred from the inactive and retired states as necessary (transitions 2 and 4 respectively in figure 5.15).

The active and inactive weights have nonzero eligibility values, so it seems that ξ will need to be modified for them at each time step according to equation 5.21. However, FOX only keeps the *active* weight values up-to-date, because they are the only ones that contribute to the CMAC output. An inactive weight and its corresponding ξ are not modified until the weight becomes active or retired, whereupon they are updated to their correct values. Active weights must always have their correct value, because they contribute to the CMAC output. Retired weights must also always have their correct value, because no other information about them is stored by FOX. In a typical CMAC most weights will be retired.

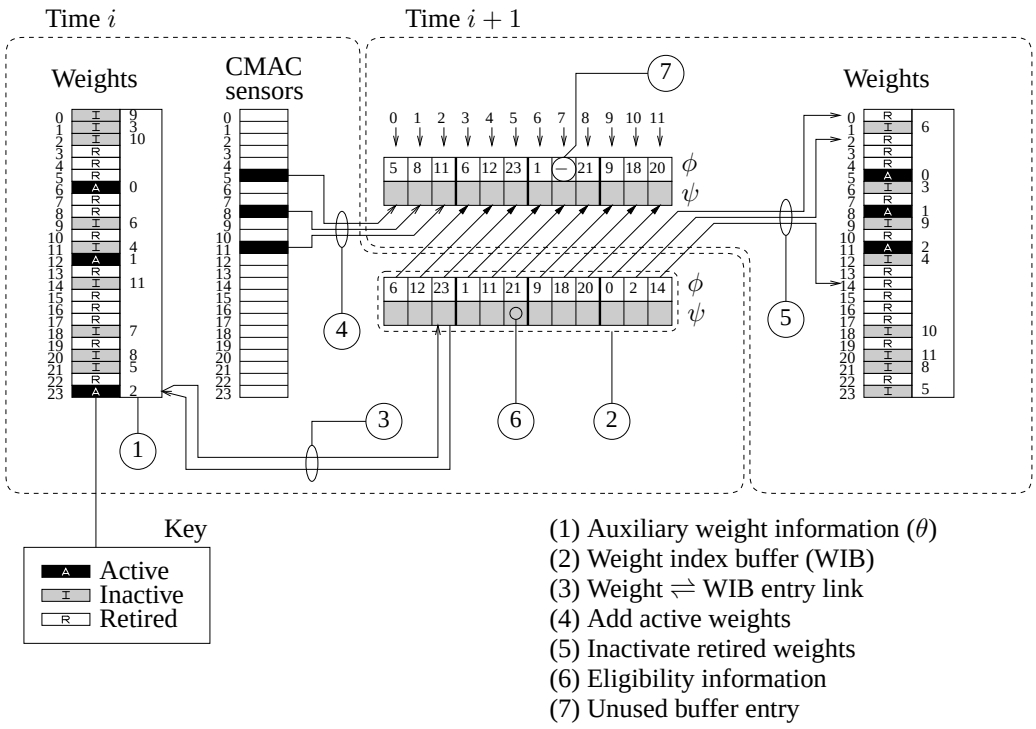


Figure 5.16: *Operation of the weight index buffer (WIB) for* $n_s = 24$, $n_a = 3$ *and* $\sigma = 4$.

The indexes of the inactive weights (indexes into $\mathbf{s}$) are stored in a shift register called the "weight index buffer" (WIB). Figure 5.16 demonstrates the operation of the WIB from time step i to time step $i+1$. The WIB has room for σ groups of n_a weight indexes. Each WIB entry stores an index into the weight table and some information about the eligibility of that weight. All weights not represented in the WIB are assumed to have zero eligibility. Each weight has an "auxiliary" number which stores that weight's position in the WIB, if any. Thus WIB entries are linked to their corresponding weights and vice versa.

During each time step the WIB is shifted (to the right in figure 5.16). The indexes of the n_a currently active weights are shifted in from the left (5,8 and 11 in figure 5.16). The n_a indexes shifted out from the right correspond to inactive weights that are σ time steps old—these weights are retired (0,2 and 14 in figure 5.16). A particular weight index cannot appear in the WIB more than once. If a weight index shifted in is already in the WIB it will be removed from its old position and that position will be marked "unused" (this happens to index 11 in figure 5.16). This will happen if one or more of the CMAC sensor values remains constant from one time step to the next.

5.7.3 Correcting inactive weights

If a weight is made inactive at time i_1 and it becomes active or retired at time i_2, its value (and the corresponding eligibility value) must be corrected to account for the interval $i_1 \ldots i_2$ during which it was not modified. This can be done efficiently for the linear F^*, as ξ will have a sum-of-exponentials decay profile. The eligibility update equation is:

$$\xi_{i+1}^{pq} \quad = \quad A\,\xi_i^{pq} + B\,\hat{\mathbf{s}}_i^{pq} \tag{5.23}$$

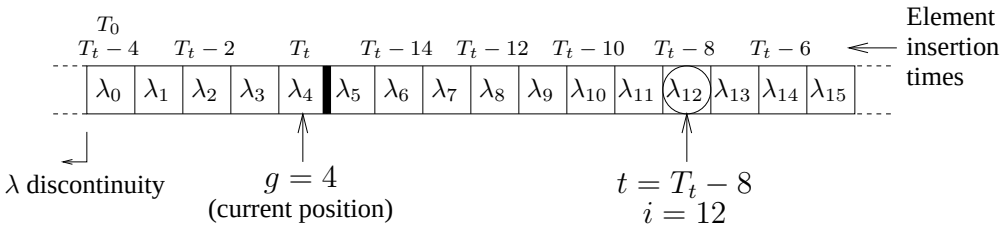


Figure 5.17: *An example trace buffer (WIB) with* $\delta = 16$.

If weight w_{pq} goes inactive at time step i_1 then $\hat{\mathbf{s}}_i^{pq} = 0$ for $i \geq i_1$, so:

$$\xi_{i+1}^{pq} \quad = \quad A\, \xi_i^{pq}, \qquad i \geq i_1 \tag{5.24}$$

thus

$$\xi_{i_1+i}^{pq} = A^i\, \xi_{i_1}^{pq}, \qquad i \geq 0 \tag{5.25}$$

If w_{pq} is made inactive (or "frozen") at time i_1, to find its true value at time i_2 the following must be computed:

$$w_{pq} \quad \leftarrow \quad w_{pq} - \alpha \sum_{i=i_1}^{i_2} e_i\, C\, \xi_i^{pq} \tag{5.26}$$

$$= \quad w_{pq} - \alpha \left[\sum_{i=i_1}^{i_2} e_i\, C\, A^{i-i_1} \right] \xi_{i_1}^{pq} \tag{5.27}$$

$$= \quad w_{pq} - \alpha\, (\lambda_{i_2} - \lambda_{i_1-1})\, A^{-i_1}\, \xi_{i_1}^{pq} \tag{5.28}$$

$$\text{where} \quad \lambda_j \quad = \quad \sum_{i=1}^{j} e_i\, C\, A^i \tag{5.29}$$

Thus if the value λ is accumulated for each time step, any inactive weight can be activated or retired with a fast calculation (equation 5.28). Values of λ must be kept for at least the last σ time steps. Note that only *one* calculation of λ has to be made for the entire system, not one per weight.

There is one remaining complication: the exponentially decreasing A^i factor in equation 5.29 causes λ to quickly converge to a steady value. As time increases, equation 5.28 multiplies an exponentially increasing value (A^{-i_1}) by an exponentially decreasing difference $(\lambda_{i_2} - \lambda_{i_1})$. The result will quickly lose numerical precision.

This can be rectified by periodically resetting λ to zero (as well as resetting the factor A^i to the identity matrix) and then accounting for the discontinuities. This idea is implemented in the "trace" algorithms shown in table 5.3. The TRACERESET procedure is first called to reset some internal variables. Then the TRACENEXT procedure is called for each time step to supply a sequence of scalar error values $e_1, e_2, \ldots e_i$. The TRACEDELTA function allows computation of the following sum without loss of numerical precision:

$$\Delta_t = \sum_{i=t}^{T_t} e_i\, C\, A^{i-t} \tag{5.30}$$

THE TRACE ALGORITHM

Parameters

n_y — Eligibility vector size (integer ≥ 1).
A, C — Matrices, of size $n_y \times n_y$ and $1 \times n_y$ respectively.
δ — Buffer size (integer ≥ 2). This equals $\sigma + 2$

Internal state variables

$\lambda_0 \ldots \lambda_{\delta-1}$ — The buffer, an array of δ vectors (size $1 \times n_y$).
T_t — Current time step.
g — The position of the current time step in the buffer.
T_0 — The time step at λ_0.
Λ — The accumulated value of: $e\, CA^i$ (size $1 \times n_y$).
Γ — The current value of: CA^g (size $1 \times n_y$).

ALGORITHM: TRACERESET — Reset to time 0.

$\Rightarrow$ for $i \leftarrow 0 \ldots (\delta - 1) : \lambda_i \leftarrow 0$
$T_t \leftarrow -1, \quad g \leftarrow -1$ —next time / buffer position is 0
$T_0 \leftarrow 0, \quad \Lambda \leftarrow 0$
$\Gamma \leftarrow C$ —Get $\Gamma = CA^0$

ALGORITHM: TRACENEXT — Set the next error value e, this is called first for time 0.

Input: e (scalar)

$\Rightarrow$ $T_t \leftarrow T_t + 1, \quad g \leftarrow g + 1$
if $g \geq \delta$ then : $g \leftarrow 0$
if $g = 0$ then : $\quad T_0 \leftarrow T_t, \quad \Lambda \leftarrow 0, \quad \Gamma \leftarrow C$
$\Lambda \leftarrow \Lambda + e\,\Gamma, \quad \lambda_g \leftarrow \Lambda, \quad \Gamma \leftarrow \Gamma\, A$ —Get $\Gamma = CA^g$ for the next step.

ALGORITHM: TRACEDELTA — Compute Δ_t, assuming $0 \leq t \leq T_t$ and $t \geq T_t - \delta + 2$.

Input: t (scalar)
Output: Δ_t (vector, size $1 \times n_y$)

$\Rightarrow$ ensure that $t \leq T_t$ and $t \geq 0$
$i \leftarrow t - T_0$ —i is the buffer position for time t
if $i < 0$ then
    $i \leftarrow i + \delta$
    ensure that $i > g$ —now i is in the range $0 \ldots \delta - 1$
ensure that $i \neq g + 1$
if $i \leq g$ then
    if $i > 0$ then
        $\Delta_t \leftarrow (\lambda_g - \lambda_{i-1})A^{-i}$
    else
        $\Delta_t \leftarrow \lambda_g\, A^{-i}$
else
    $\Delta_t \leftarrow (\lambda_{\delta-1} - \lambda_{i-1})A^{-i} + \lambda_g\, A^{T_0 - t}$

Table 5.3: *The "trace" algorithms for computing weight updates.*

where T_t is the "current" time, in terms of which the weight update equation is:

$$w_{pq} \quad \leftarrow \quad w_{pq} - \alpha \, \Delta_{i_1} \, \xi_{i_1}^{pq} \tag{5.31}$$

The trace algorithm maintains a buffer of δ past λ values (where $\delta = \sigma + 2$), as shown in figure 5.17. Here g is the index of the λ value for the "current" time T_t. Each call to TRACENEXT increases g and deposits a new λ value (g wraps around to zero at the end of the buffer). The buffer size limits the value of $T_t - t$ to at most $\delta - 2$, or σ. Inspection of the TRACENEXT procedure shows that the λ values are set as follows:

$$\text{for } 0 \le i \le g: \qquad \lambda_i \quad = \quad \sum_{j=0}^{i} e_{T_0+j} \, C \, A^j \tag{5.32}$$

$$\text{for } i > g: \qquad \lambda_i \quad = \quad \sum_{j=0}^{i} e_{T_0-\delta+j} \, C \, A^j \tag{5.33}$$

$$\tag{5.34}$$

Precision can be maintained because A^i never gets too small, as the exponent i never exceeds $\delta - 1$: each time the position g wraps around to zero the exponent on A^i is reset to zero. To calculate equation 5.30 the TRACEDELTA function combines λ terms with the appropriate scaling factors to compensate for the effect of the λ_0–$\lambda_{\delta-1}$ discontinuity. It is easy to show[3] that the TRACEDELTA function gives equation 5.30 for all valid values of t.

With this trace algorithm there is at most one λ-reset discontinuity between any valid time t and the current time T_t. A more general algorithm would allow the λ-reset interval to be selected independently of the buffer length δ. However, this introduces a lot of extra complexity (as more discontinuities must be compensated for) for little gain.

There remains the question of how σ (and therefore δ) should be chosen. This is analysed fully in Appendix F. The guideline is that A^σ should be "small but not too small". σ should be large enough such that an eligibility ξ decaying for σ time steps is small enough that its weight can be retired. σ should be small enough that the λ values stored in the buffer retain enough variation so that their differences do not lose too much numerical precision.

A complicating factor is that A is a matrix, not a scalar. It turns out that the determining factor for numerical precision is the eigenvalue of A with the smallest magnitude—call it q (if A is a scalar then $q = A$). The maximum number of decimal digits of precision that can be lost when calculating Δ_t is

$$\text{maximum decimal precision lost} \quad = \quad -\sigma \frac{\ln q}{\ln 10} \tag{5.35}$$

A proof of this is presented in Appendix F. A practical compromise is to select σ smaller than

$$\text{maximum } \sigma \quad \approx \quad -2 \frac{\ln 10}{\ln q} \approx \frac{-4.605}{\ln q} \tag{5.36}$$

This results in at most two lost digits of decimal precision, and (if the eigenvalues of A are all close enough to q) the eligibility will decay to at most approximately 1% of its starting value.

5.7.4 Summary of the algorithm

The FOX algorithm for a single output ($n_x = 1$) is given in table 5.4. The WIB is stored in the ϕ and ψ arrays. ϕ stores the indexes of the active and inactive weights. Each entry of ψ stores the eligibility

[3] The proof will be left as an exercise for the reader.

THE FOX ALGORITHM

Parameters

All CMAC parameters (see table 3.1), but assume $n_x = 1$.

σ : The size of the internal buffer

Internal state variables

$\theta[0] \ldots \theta[n_w - 1]$ — An array of n_w auxiliary numbers, one for each weight.

$\phi[0] \ldots \phi[n_a\sigma - 1]$ — An array of $n_a\sigma$ weight indexes (scalar).

$\psi[0] \ldots \psi[n_a\sigma - 1]$ — An array of $n_a\sigma$ eligibility vectors (size $n_y \times 1$).

h — The current buffer head position (scalar).

T_f — The current time step (scalar).

ALGORITHM: FOXRESET — Reset to time 0.

$\Rightarrow$ if (this is not the first time FOXRESET has been called) then:

$T_f \leftarrow T_f + 1, \quad h \leftarrow h + n_a$

if $h \geq n_a\sigma$ then : $h \leftarrow 0$

for $i \leftarrow 0 \ldots (n_a\sigma - 1)$: if $\phi[i] \neq -1$ then : FOXCORRECTWEIGHT ($\phi[i]$,1)

$T_f \leftarrow -1, \quad h \leftarrow -n_a$

for $i \leftarrow 0 \ldots (n_w - 1)$: $\theta[i] = -1$

for $i \leftarrow 0 \ldots (n_a\sigma - 1)$: $\phi[i] = -1$

TRACERESET()

ALGORITHM: FOXMAP — Map CMAC inputs to output, keep track of weights.

Inputs: $y_1 \ldots y_{n_y}$ (scalars)

Output: x_1 (scalar)

$\Rightarrow$ CMACQUANTIZEANDASSOCIATE ($y_1 \ldots y_{n_y}$) —sets $\mu_1 \ldots \mu_{n_a}$

$T_f \leftarrow T_f + 1, \quad h \leftarrow h + n_a$

if $h \geq n_a\sigma$ then : $h \leftarrow 0$

for $i \leftarrow 0 \ldots (n_a - 1)$: if $\phi[h + i] \neq -1$ then : FOXCORRECTWEIGHT ($\phi[h + i]$,1)

for $i \leftarrow 0 \ldots (n_a - 1)$

FOXCORRECTWEIGHT (μ_{i+1},0) —sets ϵ

$\psi[h + i] \leftarrow B + \epsilon$ —driving part of F^* filter

$\phi[h + i] \leftarrow \mu_{i+1}$

$\theta[\mu_{i+1}] \leftarrow h + i$

$x_1 = \sum_{i=1}^{n_a} W_1[\mu_i]$

ALGORITHM: FOXCORRECTWEIGHT — Compute the correct value for weight i.

Inputs: i, old (scalars)

Output: ϵ (vector of size $n_y \times 1$)

$\Rightarrow$ if $\theta[i] < 0$ then : $\epsilon = 0$ and exit —do nothing if weight not in buffer

$\phi[\theta[i]] \leftarrow -1$ —take weight out of buffer

bpos $\leftarrow \lfloor \theta[i]/n_a \rfloor$ —find buffer timestep position

hd $\leftarrow \lfloor h/n_a \rfloor$

$t \leftarrow T_f +$ bpos $-$ hd —find time t when weight last active

if (old $= 1$ and bpos $\geq$ hd) or (old $= 0$ and bpos $>$ hd) then : $t \leftarrow t - \sigma$

TRACEDELTA (t) —sets Δ_t

$W_1[i] \leftarrow W_1[i] + \Delta_t \, \psi[\theta[i]]$

if old=0 then : $\epsilon = A^{T_f - t} \, \psi[\theta[i]]$ else $\epsilon = 0$

Table 5.4: *The FOX algorithm for a single output.*

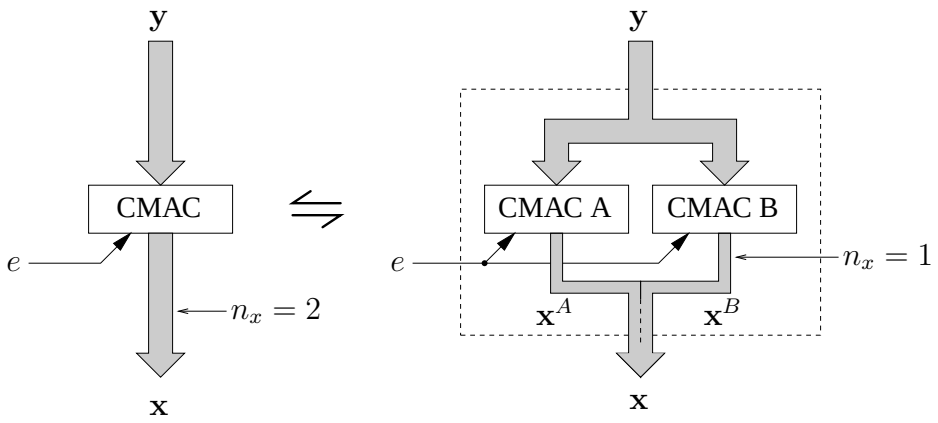


Figure 5.18: *A simple way to extend the FOX algorithm to $n_x = 2$.*

vector that the corresponding WIB weight had when it was last active. If $\phi[i] = -1$ then that WIB entry is unused. θ stores the weight auxiliary information. If $\theta[i] = -1$ then that weight is retired. This algorithm requires variant-2 CMAC hashing. This is because if two or more indexes in the WIB can point to the same weight then the eligibility computations become more complicated. This is how the FOX algorithm is used:

1. Call FOXRESET to reset the system time to 0.
2. For each system time step:
   (a) Call FOXMAP to map the input vector y to the output x.
   (b) Use the output x in the system and measure e, the resulting output error.
   (c) Call TRACENEXT ($\alpha\, e/n_a$).

The error is divided by n_a in the call to TRACENEXT to compensate for the number of AUs in the CMAC. At most n_a new weights can be made active per time step, so at most n_a weights will need to be retired per time step. Thus the total computation per time step is $\mathcal{O}(n_a)$. The FOX and TRACE algorithms can be sped up in several ways:

- In TRACEDELTA the matrix exponents (A^i) need not be calculated each time. Instead the values of A^i for $i = -\delta \ldots \delta$ can be buffered.
- In FOXCORRECTWEIGHT, if the weight to be updated was active in the last timestep (i.e. $T_t - t = 1$), as most weights usually are, then it can be updated quickly by

$$W_1[i] \leftarrow W_1[i] + (\text{last\_error\_passed\_to\_TRACENEXT})\, C\, \psi[\theta[i]] \tag{5.37}$$

These improvements are present in the FOX C++ source code (Appendix H).

Extending the FOX algorithm to $n_x > 1$ is relatively easy. One particularly simple way is shown in figure 5.18. Here a single CMAC with two outputs ($n_x = 2$) is replaced by two CMAC's with one output. The same error signal is fed to CMAC-A and CMAC-B, but the internal eligibility filters are different (reflecting the different effects of $\mathbf{x}^A$ and $\mathbf{x}^B$ on the system). Using two independent CMACs in this way is wasteful because computations common to both are duplicated. To gain full efficiency both CMACs must be combined into one algorithm. The details are left as an exercise for the reader.

5.8 More flexible error functions

Thus far the following error function has been assumed:

$$E = \sum_{i=1}^{T} \left(C\,\mathbf{y}_i - y_i^d \right)^2 \tag{5.38}$$

This will be called the *standard* error function. It has two limitations in practical systems. First, there is no constraint on the control signal $\mathbf{x}$. This can result in overtraining, as seen in Chapter 4 and Chapter 6. Second, the form of equation 5.38 does not allow multiple elements of the vector $\mathbf{y}$ to meet independent targets. These limitations will now be addressed.

5.8.1 Adding x constraints (output limiting)

As shown in the previous chapter, it will usually be necessary to limit the magnitude of $\mathbf{x}$ somehow in a practical system. Doing so means that the minimum-error trajectory from any starting point off the desired trajectory is fully specified. Here is an addition to the standard error function which directly constrains the magnitude of $\mathbf{x}$:

$$E = \underbrace{\sum_{i=1}^{T} \left(C\,\mathbf{y}_i - y_i^d \right)^2}_{\text{part 1}} + \underbrace{\frac{\beta}{2} \sum_{i=1}^{T} \mathbf{x}_i^T\,\mathbf{x}_i}_{\text{part 2}} \tag{5.39}$$

This will be called the *output limiting* error function. The weight gradient can be found using the chain rule, as before:

$$\frac{\mathrm{d}\,E}{\mathrm{d}\,w_{pq}} = \left(2 \sum_{k=2}^{T} e_k\,C\,\xi_k^{pq} \right) + \sum_{i=1}^{T-1} \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{x}_i} \cdot \frac{\mathrm{d}\,\mathbf{x}_i}{\mathrm{d}\,w_{pq}} \tag{5.40}$$

$$= \left(2 \sum_{k=2}^{T} e_k\,C\,\xi_k^{pq} \right) + \beta \sum_{i=1}^{T-1} \mathbf{x}_i^T\,\hat{\mathbf{s}}_i^{pq} \tag{5.41}$$

$$= \left(2 \sum_{k=2}^{T} e_k\,C\,\xi_k^{pq} \right) + \beta \sum_{i=1}^{T-1} \mathbf{x}_i^{(p)}\,\mathbf{s}_i^{(q)} \tag{5.42}$$

The bracketed term on the left is the standard eligibility equation arising from part 1 of the error function (i.e. equation 5.15). The term on the right arises from part 2. This results in the following weight update rule:

$$w_{pq} \leftarrow w_{pq} - \alpha \sum_{i=2}^{T} e_i\,C\,\xi_i^{pq} - \left(\beta \sum_{i=1}^{T-1} \mathbf{x}_i^{(p)}\,\mathbf{s}_i^{(q)} \right) \tag{5.43}$$

The bracketed term on the right is the addition to the FOX algorithm caused by part 2 of equation 5.39. It is equivalent to normal CMAC target training with a target of zero. Thus training with output limiting is implemented in the following way:

1. Call FOXRESET to reset the system time to 0.
2. For each system time step:

(a) Call FOXMAP to map the input vector y to the output x.

(b) Use the output x in the system and measure e, the resulting output error.

(c) Call TRACENEXT ($\alpha\, e/n_a$).

(d) Call CMACTARGETTRAIN (0, β)

5.8.2 Adding $\mathbf{x}$ derivative constraints

Sometimes limiting the magnitude of $\mathbf{x}$ is not useful because it may be required to grow arbitrarily large to perform some function in the system. In such situations $\mathbf{x}$ can be prevented from driving the system too hard by limiting the magnitude of $\mathrm{d}\mathbf{x}/\mathrm{d}t$ instead. For instance, the following error function does this with the first order numerical derivative of $\mathbf{x}$:

$$E \;=\; \gamma \sum_{i=1}^{T-1} |\mathbf{x}_{i+1} - \mathbf{x}_i|^2 \tag{5.44}$$

But in fact this error is less than perfect for implementation in FOX. Consider:

$$\frac{\mathrm{d}\, E}{\mathrm{d}\, w_{pq}} \;=\; \left[2\gamma \sum_{i=1}^{T-1} \Big(-\mathbf{x}_{k-1} + 2\mathbf{x}_k - \mathbf{x}_{k+1}\Big)^T \hat{\mathbf{s}}_i^{pq}\right] + 2\gamma \mathbf{x}_1^T \tag{5.45}$$

For a time span $i = T_1 \ldots T_2$ in which $\hat{\mathbf{s}}_i^{pq}$ is constant, changing the value of $\mathbf{x}_j$ (where $T_1 < j < T_2$) will not affect the value of $\mathrm{d}E/\mathrm{d}w_{pq}$. What this means is that $\mathrm{d}E/\mathrm{d}w_{pq}$ is only affected by discontinuities in the value of $\hat{\mathbf{s}}_i^{pq}$. This property has been found in practice to limit the effectiveness of the error function in constraining the $\mathbf{x}$ trajectory.

A more active (although somewhat heuristic) alternative is to train the CMAC output at each iteration using the CMAC output of the previous timestep as a target. This will be called *output derivative limiting*. It results in the following FOX algorithm:

1. Call FOXRESET to reset the system time to 0.
2. $\mathbf{x}_{\text{old}} \leftarrow 0$
3. For each system time step:
   (a) Call FOXMAP to map the input vector y to the output x.
   (b) Use the output x in the system and measure e, the resulting output error.
   (c) Call TRACENEXT ($\alpha\, e/n_a$).
   (d) Call CMACTARGETTRAIN ($\mathbf{x}_{\text{old}}$, γ)
   (e) $\mathbf{x}_{\text{old}} \leftarrow \mathbf{x}$

5.8.3 Adding extra $\mathbf{y}$ constraints

Consider the case where there are two seperate $\mathbf{y}$ constraints:

$$E \;=\; \sum_{i=1}^{T} \Big(\underbrace{C_A\, \mathbf{y}_i - y_i^{dA}}_{=\, e_i^A}\Big)^2 + \sum_{i=1}^{T} \Big(\underbrace{C_B\, \mathbf{y}_i - y_i^{dB}}_{=\, e_i^B}\Big)^2 \tag{5.46}$$

This leads to the following weight update equation:

$$w_{pq} \leftarrow w_{pq} - \alpha_A \sum_{i=2}^{T} e_i^A \, C_A \, \xi_i^{pq} - \alpha_B \sum_{i=2}^{T} e_i^B \, C_B \, \xi_i^{pq} \tag{5.47}$$

There are now two different error signals (e^A and e^B) and two different eligibility profiles ($C_A\xi$ and $C_B\xi$). Thus the FOX algorithm's data structures must effectively be duplicated, i.e. two WIB buffers and two ψ eligibility vector buffers are required. The resulting algorithm is structurally similar to table 5.4 (its derivation is relatively simple and is left as an exercise for the reader). As with the multiple-output case this may be easily implemented using two independent CMACs, except that in this case both CMACs must reference the *same* weight table. This argument may be extended to more than two constraints.

5.9 Special training situations

Some situations require special training procedures or special handling of the eligibility values.

5.9.1 Disturbances

If the system F is disturbed then the training process will be degraded. This is because the FOX model of the system does not encompass any mechanism that leads to the disturbance—in other words, it is unexpected. A disturbance can be an external force or noise signal that is applied unexpectedly to the system. It can also be something like a change in the control target, reference trajectory or system dynamics. There are three main ways to deal with disturbances:

1. Ignore them. This will result in some incorrect training, but if the disturbances are infrequent then this may be acceptable.

2. Anticipate the disturbances by feeding the appropriate sensor information to the CMAC. The sensor information must be sufficient to associate the cause of the disturbance with its effect on the system. Ideally this sensor information should anticipate the disturbance by some appropriate amount of time, so that FOX can *predict* it and optimally correct for it.

3. Whenever a disturbance occurrs, reset all the weight eligibilities to zero. This is done because the memory of the past context (encoded in the eligibilities) becomes irrelevant for predicting cause and effect after the disturbance has occurred. To reset all eligibilities to zero the FOXRESET function is called, which in turn will call FOXCORRECTWEIGHT for all weights in the WIB. Detection of controller-imposed disturbances (such as a change in the reference value) is easy. Detection of genuinely unpredictable external disturbances can be difficult, and will usually require some extra controller subsystems.

5.9.2 Switched CMACs

Consider what happens when a CMAC output can be switched in or out, so that sometimes it has no influence on the system. An example is shown in figure 5.19, which shows two CMACs multiplexed together to make a single controller. The eligibility profile should be set to zero when the CMAC output has no effect, because an impulse in the CMAC output would produce no change in the measured error. The normal eligibility profile can be used when the CMAC is switched in. The easiest way to implement this is to switch the eligibility filter driving force on and off as the CMAC is switched in and out. To do this in the FOXMAP algorithm the line

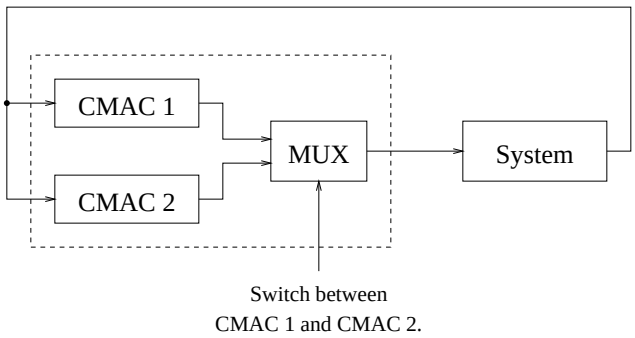


Figure 5.19: *A controller made from two CMACs multiplexed together.*

$$\psi[h+i] \leftarrow B + \epsilon$$

should be replaced with

if (CMAC is switched in) then
$\psi[h+i] \leftarrow B + \epsilon$
else
$\psi[h+i] \leftarrow \epsilon$

5.9.3 Timer control

The CMAC output can be used as a time delay (or threshold) that triggers the transition from one control state into another. In this case the CMAC output only influences the system during the instant that it triggers the state change. Therefore this point should be the only one that has a nonzero eligibility profile. This can be achieved using a modification to the FOXMAP algorithm similar to that mentioned above. The (nonzero) eligibility profile will probably have to be determined heuristically due to the nonlinearity of the CMAC output's effect.

5.10 The continuous version

Thus far discrete time systems have been described. The development applies equally to continuous time systems. For example, the continuous eligibility and weight update equations are:

$$\dot{\mathbf{y}}(t) = A\,\mathbf{y}(t) + B\,\mathbf{x}(t) \tag{5.48}$$

$$\xi^{pq}(0) = 0 \tag{5.49}$$

$$\dot{\xi}^{pq}(t) = A\,\xi^{pq}(t) + B\,\hat{\mathbf{s}}^{pq}(t) \tag{5.50}$$

$$w_{pq} \leftarrow w_{pq} - \alpha \int_0^T e(t)\,C\,\xi^{pq}(t)\,\mathrm{d}t \tag{5.51}$$

But note that for equivalent discrete and continuous systems, the A matrices are different. No further consideration will be given to continuous time systems.

5.11 Related methods

Other authors have used similar approaches for incorporating eligibility parameters in to the CMAC.

Lin and Kim [67] implement Barto's adaptive critic [10] architecture using a CMAC instead of a simple lookup table. Each CMAC weight had an associated scalar eligibility value, but no guidance was given for choosing the eligibility dynamics and a naïve eligibility update approach was used. In [68] Lin and Kim provided loose guidelines for selecting scalar eligibility dynamics, but no connection between the eligibility dynamics and the controlled system dynamics was made.

Hu and Fellman [48] implemented a "state history queue" (SHQ) for improving the speed of the simple table lookup scheme used in Barto's adaptive critic [10]. Each "box" in the adaptive critic has an associated eligibility value. Their approach has some similarities to the weight index buffer technique used in FOX (it eliminates computations for temporally insignificant weights), but it does not achieve as high a speed-up, as they process the eligibility values of *all* weights in the SHQ whereas only retired and re-activated weight eligibilities are processed in FOX's WIB.

No other authors have considered the problem of parameter selection or implementation speed for *vector* eligibilities.

5.12 Conclusion

The FOX algorithm achieves optimal control using a CMAC to learn the best control actions. FOX associates an eligibility value with each weight, which allows the information in the error signal to be used to the best effect by altering a much larger group of weights than with the CMAC. In contrast to other reinforcement learning systems, the eligibility values can be vectors (not just scalars) and their update equations can be optimally selected to match the dynamics of the controlled system. FOX maintains auxiliary "trace" information about the error signal, which can be computed cheaply as a substitute for the expensive computation of eligibility values for all weights. FOX's speed is thus virtually independent of the CMAC size or the system size.

The FOX algorithm can be regarded as an extension of the feedback-error control approach. It has been shown that FBE by itself may not be able to control systems where the control actions do not have an immediately observable effect (systems having a non-zero-order impulse response). This will be experimentally verified in the next chapter. With FBE, explicit reference trajectories must be generated to suit different situations. In contrast FOX will automatically find an error-minimizing trajectory that approaches a single reference path in a well defined way. FBE uses the same signal for feedback and error purposes, whereas FOX allows independent choice of an internal feedback controller and error signal. This gives the designer more flexibility.

The FOX algorithm's synapse eligibility make it an even better model of the cerebellum than the standard CMAC, as cerebellar Purkinje cells also have an "eligibility" property (see section 2.7). It is possible that FOX can teach us something about how Purkinje synapses operate. For example, perhaps the eligibility profile of the Purkinje cell synapse correlates somehow with the dynamics of the motor region being modulated by that cell. Perhaps the dynamics of such motor regions are sufficiently damped that they can be successfully modeled by a low order eligibilty profile. However, these arguments (by functional analogy) are not very strong, and there has been no supporting evidence presented to date. In any case, the detailed operation of the biological motor control system is still not fully understood.

The next chapter will describe some design methodologies for FOX-based systems, and will look at how successful FOX is in practice.

Chapter 6

Testing the FOX controller

6.1 Introduction

This chapter describes how to design FOX-based control systems. Various aspects of the design process are investigated, such as how to measure a system impulse response, how to construct the FOX eligibility profile model using this information, various eligibility profile modeling problems (and their solutions), and ways to cope with nonlinear systems.

Experiments were performed on one simulated system and three physical systems: a simulated gantry crane, a real gantry crane, an inverted pendulum, and a track-following wheeled robot. As each one of these experiments is described below, some new elements are introduced into the design process. In each hardware experiment the control algorithm was performed inside a real-time control process running under the Linux operating system on a Pentium class PC. The implementation details of this software system, which is a sophisticated and general purpose real-time controller, are given in Appendix K.

6.2 Design technique

This section describes the steps required to control a system using FOX.

6.2.1 Step 1: Feedback control

Many uncompensated systems will have unacceptable impulse responses, i.e. responses that can not be easily modeled as eligibility profiles. Some typical problems with an uncompensated impulse response are:

- It may not decay to zero (i.e. the system never reaches its reference value).
- It may take a long time to decay to zero (i.e. the system takes a long time to reach its reference).
- It may be too complicated, in the sense that a high order linear system model would be required for sufficient accuracy.
- It may be too nonlinear, in the sense that a linear model would not encompass some essential system dynamics.

Error function name	Equation	Learning rates
Standard	$E = \sum_{i=1}^{T} \left(C\,\mathbf{y}_i - y_i^d \right)^2$	α
Overshoot	$E = \sum_{i=1}^{T} \left(e_i' \right)^2$ (e_i' is defined in section 6.2.2)	α_1, α_2
Output limiting	$E = \sum_{i=1}^{T} \left(C\,\mathbf{y}_i - y_i^d \right)^2 + \frac{\beta}{2} \sum_{i=1}^{T} \mathbf{x}_i^T\,\mathbf{x}_i$	α, β
Output derivative limiting	(see section 5.8.2)	α, γ

Table 6.1: *Some common error functions and their learning rate parameters. Note that, except for the overshoot case, α is the main learning rate.*

- It may be too oscillatory.

An "internal" feedback controller (IFC) is added to the system to correct these problems (it is equivalent to the FBE feedback path). For mechanical systems it is usually some kind of position-reference linear controller. The IFC is included in the system block F along with the original uncompensated system. The IFC embodies some of the "expert" knowledge of how the system should be controlled.

Selecting the IFC usually involves a tradeoff. A simple IFC may result in a complicated impulse response that will be hard to model. The impulse response can be made simpler at the expense of creating a more complicated IFC. But this may limit the ability of FOX to adapt to changing system parameters, or prevent it from accurately compensating for system nonlinearities. A balanced approach is required: select a "minimal" IFC that just covers the above requirements, so FOX can still discover the system dynamics for itself from the resulting eligibility profiles.

It will be shown in section 6.6 that an impulse response that does not decay to zero can be modeled by an eligibility profile that does. This is not too surprising when it is considered that an eligibility value measures the influence that a given weight has had on the system. In general, as a system operates *any* eligibility should decay to zero because a weight's influence on the system will become less important as other weights gain control (i.e. as more recent control outputs direct the system along new trajectories).

6.2.2 Step 2: Select error function

The error function determines how the trained system will behave, so it must be selected with care. Usually several training experiments will be necessary before a satisfactory error function is obtained. The error functions that have been described thus far, and their learning rate parameters, are shown in table 6.1.

Table 6.1 introduces a heuristic technique called "overshoot error", which is sometimes useful. This is similar to the standard error function, except that the error signal e_i is redefined to be:

$$e_i' = \begin{cases} \alpha_1 \left(C\,\mathbf{y}_i - y_i^d \right) & \text{if } \left| C\,\mathbf{y}_i - y_i^d \right| - \left| C\,\mathbf{y}_{i-1} - y_{i-1}^d \right| < 0 \\ \alpha_2 \left(C\,\mathbf{y}_i - y_i^d \right) & \text{otherwise} \end{cases} \tag{6.1}$$

where typically α_2 is 5–100 times larger than α_1. If the reference error reduces over time then this is just the standard error function with learning rate α_1. If the error increases (for example if the system

overshoots its reference) then the learning rate is increased to α_2. This penalizes reference overshoots, requiring FOX to avoid them by applying a pre-emptive "braking force".

Combinations of the error functions in table 6.1 are possible, such as overshoot training plus output limiting. In fact, some amount of output limiting (or output derivative limiting) will usually be required in a practical system. The output limiting learning rate must be selected high enough to prevent over-training, but low enough so that FOX is still capable of controlling the system.

6.2.3 Step 3: Measure eligibility profiles

Measurements must be made on the system to determine the eligibility profiles. This is done as follows:

1. Set the system reference to zero.
2. Apply an impulse to each component of $\mathbf{x}$ in turn.
3. For each component, the measured error is the ideal eligibility profile.

Impulses are hard to apply in practice. A more practical method for linear systems is to apply a step function, then the impulse response will be the derivative of the measured error.

6.2.4 Step 4: Synthesize A,B,C

The impulse responses are used to construct the eligibility profiles (the linear system model F^* given by the matrices A,B and C). This model does not need to be perfect, as long as the modeled eligibility profiles are sufficiently close to the measured impulse responses. There are three main modeling techniques:

1. Theoretical: A known system model is used to obtain the form of A,B and C. Some parameters are adjusted to make the model fit the measured impulse response.
2. Algorithmic: The matrices A,B and C are synthesized directly from the measured impulse response. An algorithm to do this was not created, see section 8.1.
3. Heuristic: A generic first, second or third order model is tweaked until it matches the measured impulse response. This was the method used most often in practice.

It is sometimes possible to reduce the *order* of the model with little loss in accuracy. This occurs if at least one eigenvalue in the system matrix A is particularly small, so its contribution to the output decays quickly. The eigenvalue/eigenvector can be deleted from the eigen-expansion of A, and B and C are correspondingly modified. A procedure for doing this is given in Appendix G. An additional benefit of reducing the model order is that the FOX trace precision is better if the remaining eigenvalues are closer together.

6.2.5 Step 5: Select σ

The WIB buffer size σ must be large enough that the eligibility profile is not truncated too much, and small enough that numerical imprecision does not manifest itself in the trace calculations. A full analysis of the problem and a method for selecting σ is given in Appendix F. A guideline is too choose σ to be the number of time steps it takes for the eligibility profile to decay to 1% of its maximum value, as long as this does not conflict with trace precision requirements for the floating point data type being used.

6.2.6 Step 6: Select the CMAC inputs

This must be done so that there is just enough sensor information to uniquely identify each point on all useful system trajectories. Less sensors will result in training correlation between two or more parts of the trajectory. More sensors can increase the CMAC noise factor, although this depends on the nature of the CMAC input. It can be useful to use CMAC sensors which are pre-processed versions of the systems sensors, to reduce the required number of sensors without compromising the uniqueness requirement.

The time t can be used as a CMAC input in a feed-forward system. This has the advantage that the CMAC input is guaranteed to uniquely identify all input space points on the desired trajectory. However the system must be guaranteed to start from the same state at each training run (time $t = 0$) and have the same desired trajectory each time. Because there is no input space overlap, learning will have to occur separately at all times on the desired trajectory (even if that trajectory is periodic, with multiple identical parts).

6.2.7 Step 7: Testing

The system is operated and the FOX controller is trained. Training will only occur in those regions of the workspace that are actually seen. Thus the system reference should repeatedly traverse all the potentially useful paths during training.

The various learning rates must be adjusted to achieve stability. A higher learning rate results in faster learning, but can cause instabilities and training interference. The proportions of the main reference learning rate and the output limiting (or overshoot) learning rates need to be adjusted to get the desired "optimal" performance.

6.3 System approximation requirements

This section will give some insight into the question: how close must the approximation F^* be to the real system F for FOX training to succeed? If the weight changes for a training iteration are selected to follow the gradient $\mathrm{d}E/\mathrm{d}w_{ij}$ then the control signals $\mathbf{x}_i$ will follow the gradient $\mathrm{d}E/\mathrm{d}\mathbf{x}_i$. The backwards propagation rule for finding the $\mathrm{d}E/\mathrm{d}\mathbf{x}_i$ values can be easily derived (refer to Appendix B):

$$\frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_i} = \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_{i+1}} \cdot A + 2\,e_i\,C \tag{6.2}$$

$$\frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{x}_i} = \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_{i+1}} \cdot B \tag{6.3}$$

These two equations define an ordinary linear filter, with e_i as the input and $\mathrm{d}E/\mathrm{d}\mathbf{x}_i$ as the output, that operates backwards in time. This filter is not the quite the same as the system F^*, although both have the same system matrix A. Assume for the moment that F is linear. Let $\nabla_i = \mathrm{d}E/\mathrm{d}\mathbf{x}_i$ where the A,B and C matrices of the original system F are used. Let ∇_i^* be the output generated when the approximation F^* is used. Training convergence requires that

$$\sum_{i=1}^{T} \nabla_i \cdot \nabla_i^* > 0 \tag{6.4}$$

(see Appendix A.3 for the reason why). In hand-waving terms this means that ∇_i and ∇_i^* must be "mostly of the same sign". This is demonstrated in figure 6.1. From this it seems that a large amount of

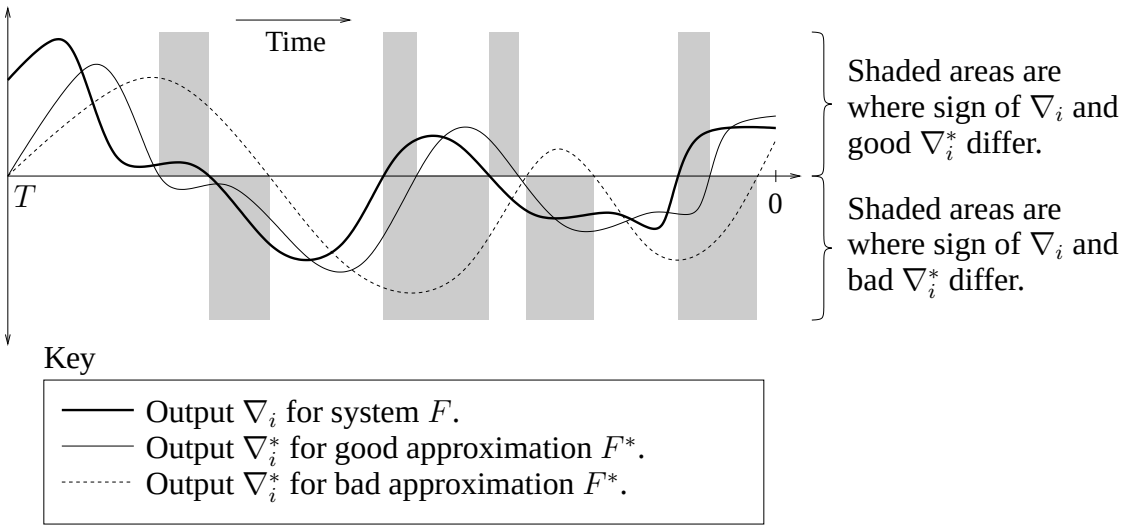


Figure 6.1: *Good and bad ∇_i^* examples. If F^* is a good model of F then ∇_i^* will track ∇_i more closely. The good ∇_i^* has less difference in sign from ∇_i.*

deviation between F and F^* would be allowable—this is confirmed by experiment later in this chapter (but obviously the closer F^* is to F the better the chance of successful training).

This fact allows a degree of confidence that FOX can control nonlinear systems. Consider the case where F^* is constructed to match a local linear approximation of a nonlinear system. As long as the linear approximations at all useful points of the state space are within the bounds of the allowable F^* deviation then training should succeed.

Notice that the exact form of the input e_i is a factor in determining if equation 6.4 is true. This fact makes it difficult to find a more formal condition for training convergence.

6.4 Testing: Simulation

The design and operation of a FOX based controller will now be illustrated using a simulated gantry crane control system. A variety of design and training strategies will be presented.

Figure 6.2a shows a simple linear fourth order model of a gantry crane (mass m_1) with a load (mass m_2) suspended on a cable (represented by a spring with spring constant 1). The FOX controller structure is shown in figure 6.2b. The crane and load are at positions p_1 and p_2. A force f is applied to the crane to get the load position equal to a reference value. The equations of motion are:

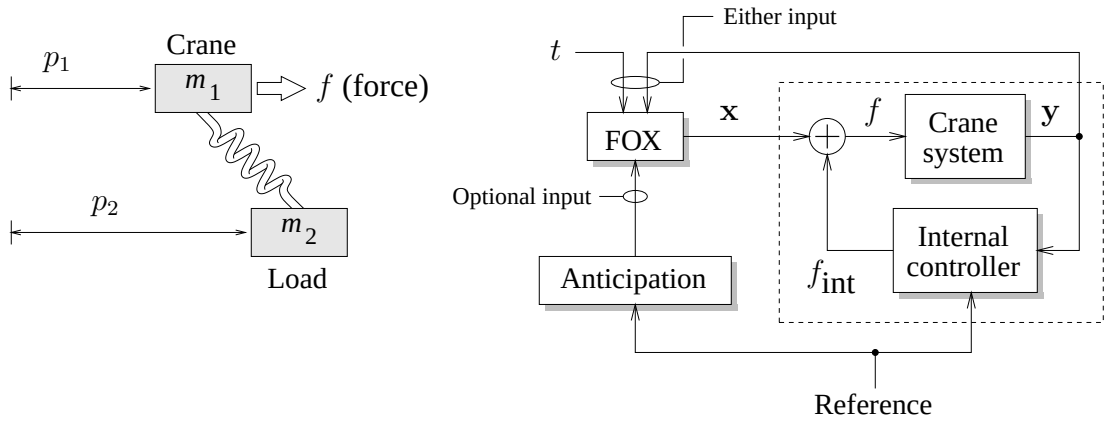
$$\ddot{p_1} = \frac{(p_2 - p_1 + f)}{m_1} \tag{6.5}$$

$$\ddot{p_2} = \frac{(p_1 - p_2)}{m_2} \tag{6.6}$$

$$f = \mathbf{x} + f_{\text{int}} \tag{6.7}$$

$$f_{\text{int}} = k_1\, p_1 + k_2\, p_2 + k_3\, \dot{p_1} + k_4\, \dot{p_2} \tag{6.8}$$

Note that the system includes an "internal" PD controller which adds a force f_{int} to $\mathbf{x}$. The system state



(a) (b)

Figure 6.2: *(a) Simple linear gantry crane model. (b) FOX controller structure for controlling this system.*

vector is

$$\mathbf{y} = \begin{bmatrix} p_1 \\ p_2 \\ \dot{p_1} \\ \dot{p_2} \end{bmatrix} \tag{6.9}$$

The following parameters were used in all the experiments:

$$m_1 = 5 \qquad m_2 = 1 \tag{6.10}$$

$$k_1 = -3 \qquad k_2 = 2 \qquad k_3 = -15 \qquad k_4 = 10 \tag{6.11}$$

The resulting system matrices are (using an Euler approximation with time step $h = 0.1$):

$$A = \begin{bmatrix} 1 & 0 & 0.1 & 0 \\ 0 & 1 & 0 & 0.1 \\ -0.08 & 0.06 & 0.7 & 0.2 \\ 0.1 & -0.1 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 \\ 0 \\ 0.02 \\ 0 \end{bmatrix} \tag{6.12}$$

Note that A and B represent both the gantry crane system and the internal controller.

6.4.1 Standard error function with the fourth order model

The second step of the FOX design process is to define an error function:

$$\text{error} = e = p_2 - \text{reference} \tag{6.13}$$

$$= C\,\mathbf{y} - \text{reference} \tag{6.14}$$

$$\text{where} \quad C = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \tag{6.15}$$

Then the system impulse response (desired eligibility profile) must be determined. This is particularly easy because the A and B matrices are already known. The impulse response, shown in figure 6.3, is

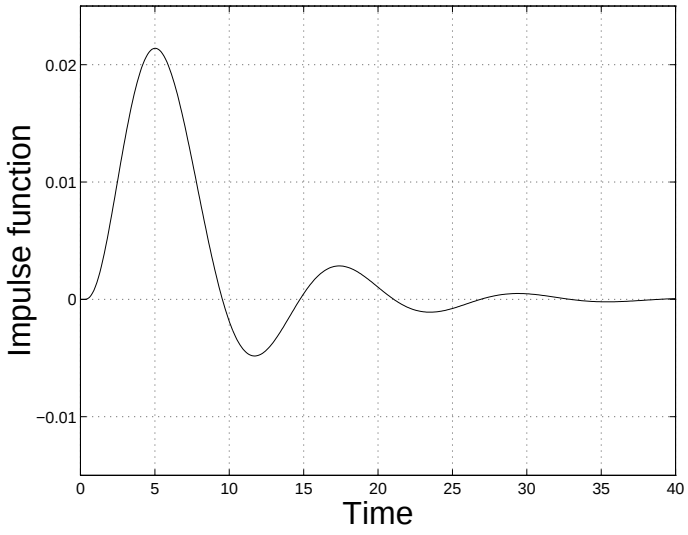


Figure 6.3: *Impulse response of the gantry crane system (including its internal controller).*

System time step (h)	0.1
Total time steps per iteration (T)	400
Total number of iterations	20000
CMAC input resolution	400
CMAC input minimum	0
CMAC input maximum	40
CMAC number of weights	100000
CMAC n_a	20
Main learning rate (α)	0.05

Table 6.2: *The CMAC parameters that were used for most of the simulated gantry crane experiments.*

$f(i) = CA^iB$. In this case the system matrices A and B will be used unchanged in the FOX controller, so no approximation step is necessary (in other words, the full fourth order eligibility model is used). The history size σ is set to 350, because the impulse response has almost fully decayed after 35 seconds. The consequences of this choice will be explored below. Finally, the CMAC input is selected to be the time t ($t = ih$ where i is the time step number). This is only possible because the simulation starts at the same state and has the same reference (which is a function of t) in each iteration. The other CMAC and simulation parameters for the gantry crane experiments are shown in table 6.2.

Figure 6.4 shows how the error improves steadily with the number of training iterations. The total error in this case was computed by:

$$\text{total error} = E = \sqrt{\sum_{i=1}^{T} h\, e_i^2} \tag{6.16}$$

Figure 6.5 shows the reference signal and the trajectories that result after training is completed. The load

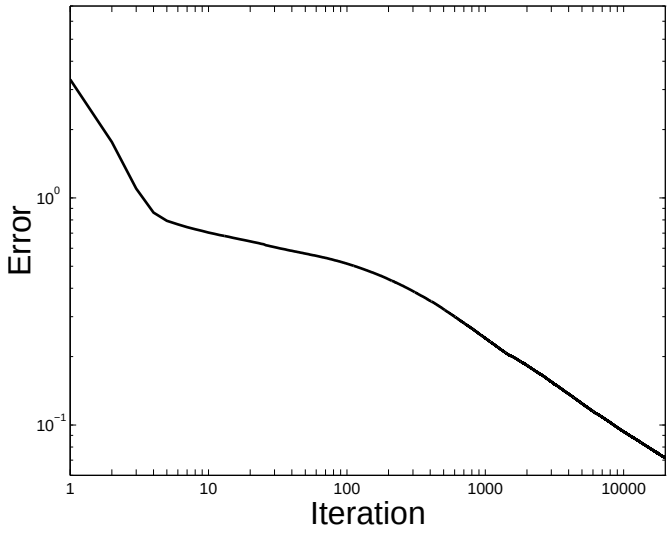


Figure 6.4: *Gantry crane controller simulation: Reference error improvement with number of iterations. The standard FOX error function with a fourth order eligibility model is used.*

position p_2 is almost exactly equal to the reference. There is nothing remarkable about this result—the FOX eligibility profile is exact and the system is linear and so convergence is ensured.

The fourth order eligibility model in this simulation required 320 bits of floating point precision[1] to achieve sufficient accuracy. By comparison, the IEEE single and double precision floating point numbers have 24 and 48 bits of precision respectively. A C++ high precision floating point number class was used instead of the usual C `float` or `double` data types. As a result the 20000 iterations took two days to compute on a Pentium II 300MHz machine. Needless to say, such extraordinary numerical precision is impractical.

The eigenvalues of A can be examined to see why such accuracy is required:

$$\text{Eigenvalues of } A = \begin{bmatrix} 0.9847 + 0.052i \\ 0.9847 - 0.052i \\ 0.9718 \\ 0.7589 \end{bmatrix} \tag{6.17}$$

The smallest eigenvalue is 0.7589. Thus from equation 5.35 at least 42 decimal digits of precision are required. The 160 fraction bits give about 48 decimal digits precision, just a bit more than is required. If a standard IEEE floating point type has to be used then the only way to make it work is to use a small history size - but small history values will chop off most of the eligibility profile.

[1] Fixed point numbers were used with 160 bits in the integer part and 160 bits in the fractional part. The integer part needs the same number of bits so that all reciprocals are representable.

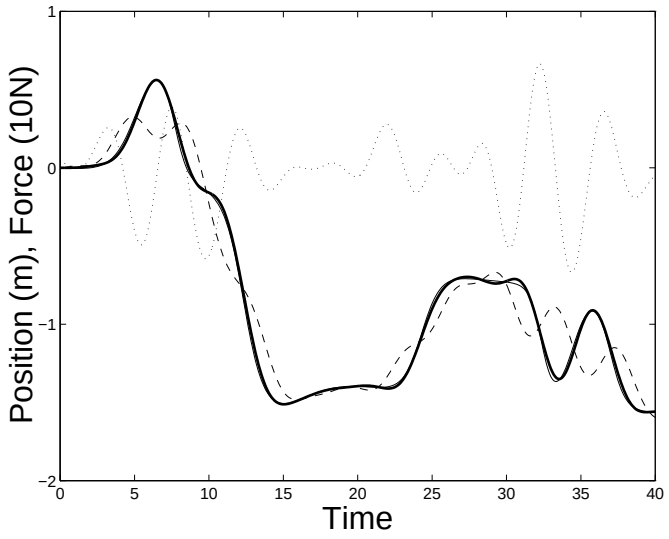


Figure 6.5: *Gantry crane controller simulation: The trajectories that result after training is completed (the standard FOX error function with a fourth order eligibility model is used). Key: reference (—), p_2 (▬), p_1 (– – –), $\mathbf{x}$ ($\cdots$). Note that the reference and p_2 are almost coincident.*

6.4.2 Reduced order models

In a practical system the matrix A has to be reformulated to prevent such precision problems. The technique of Appendix G was used to get reduced third and second order eligibility models:

$$A^{(3)} = \begin{bmatrix} 1 & 0 & 0.1 \\ 0.095 & 0.918 & 0.161 \\ 0.110 & -0.103 & 1.023 \end{bmatrix} \quad B^{(3)} = \begin{bmatrix} 0.012 \\ 0.002 \\ -0.009 \end{bmatrix} \quad C^{(3)} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \tag{6.18}$$

$$A^{(2)} = \begin{bmatrix} 0.739 & 0.193 \\ -0.326 & 1.230 \end{bmatrix} \quad B^{(2)} = \begin{bmatrix} -0.023 \\ -0.031 \end{bmatrix} \quad C^{(2)} = \begin{bmatrix} 0 & 1 \end{bmatrix} \tag{6.19}$$

The corresponding eligibility profiles are shown in figure 6.6. The third order model is an adequate approximation, as it is only substantially different in the first second or so. This is because the eigenvalue that was removed (0.7589) causes rapid decay compared to the others. The second order model is a bad approximation (it does not start near zero). Note that eigenvalue elimination can not give a first order approximation in this case, because the remaining complex eigenvalue will result in a complex A and B.

To get the first and second order models a hill-climbing least squares technique was used to minimize the squared error between the models and the full fourth order trajectory (figure 6.7). The resulting models are:

$$A^{(2)} = \begin{bmatrix} 1 & 0.1 \\ -0.0113 & 0.97 \end{bmatrix} \quad B^{(2)} = \begin{bmatrix} 0 \\ 0.0113 \end{bmatrix} \quad C^{(2)} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{6.20}$$

$$A^{(1)} = \begin{bmatrix} 0.9888 \end{bmatrix} \quad B^{(1)} = \begin{bmatrix} 0.1 \end{bmatrix} \quad C^{(1)} = \begin{bmatrix} 1 \end{bmatrix} \tag{6.21}$$

The second order model is not a particularly close match, but it does manage to follow the major peaks of the fourth order model. The first order model is quite dissimilar to the fourth order model. A zero-

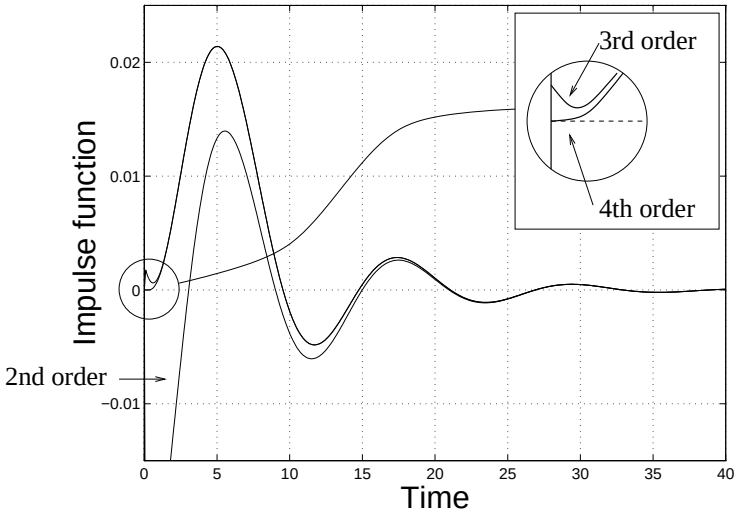


Figure 6.6: *Reduced order eligibility profiles of the gantry crane system. The fourth order function is the same as in figure 6.3. The third and second order functions have their first and second (respectively) smallest eigenvalues removed.*

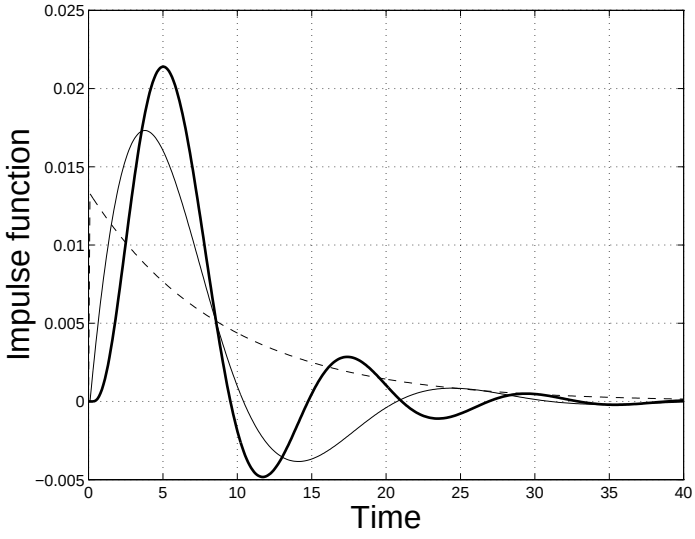


Figure 6.7: *The first and second order eligibility models for the gantry crane system, obtained by minimizing the squared error between the models and the full fourth order trajectory. Key: fourth order (—), second order (— —), first order (– – –).*

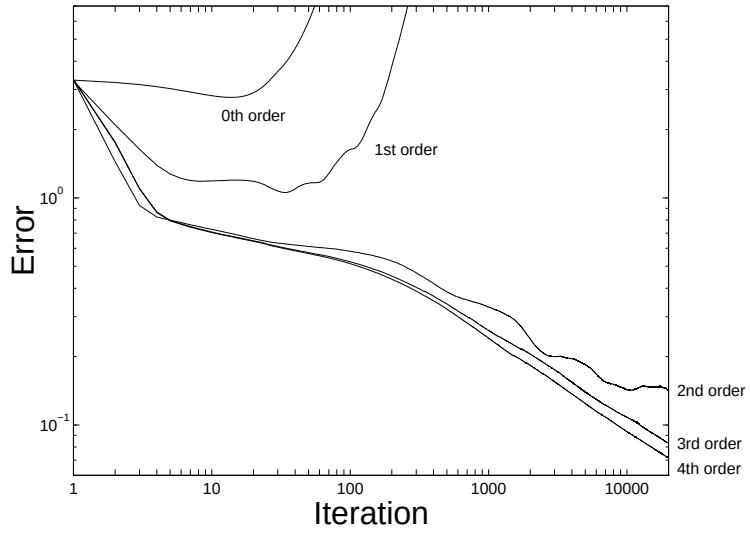


Figure 6.8: *Reference error vs number of iterations for eligibility model orders* $0 \ldots 4$.

order model is just a CMAC without any of the FOX eligibility machinery, but for this experiment it was emulated using $A = B = C = 1$ and a history buffer size of 1. A history buffer size of 300 is used for orders $1 \ldots 3$. For all computations involving the reduced order models, IEEE single precision floating point numbers with 7.2 decimal digits of precision were adequate (training times were on the order of several minutes for 20000 iterations).

How effective are the reduced order models? Figure 6.8 shows the error performance when each model is used in the FOX controller (the overshoot error function was used, with $\alpha_1 = 0$, $\alpha_2 = 0.1$). The third order model performs almost as well as the fourth, as would be expected because their eligibility profiles are so similar. The second order model also has a similar performance, though with a slightly slower and more varied convergence rate. The first and zero-order models fail to converge at all (the magnitude of $\mathbf{x}$ grows without bound). Note that the zero-order model corresponds to a standard FBE system, which means than FBE can not be successfully used in this situation. The conclusion is that the FOX eligibility profile must be roughly the same as the system's impulse response, but a large amount of variation is allowed. Other things being equal a lower order model is preferable because it requires less computation in the FOX algorithm. In all subsequent experiments the third order model will be used.

6.4.3 Training methods

Different error functions result in different training behavior. Three error functions will be considered here (in isolation and in combination). They are the standard and output limiting error functions controlled by α and β, and the overshoot error function controlled by α_1 and α_2. Figure 6.9 shows the effect of these error functions, separately and in combination, with second and third order eligibility models. A reference trajectory with a large discontinuity is used, as shown in figure 6.10.

In figure 6.9 the second order model using the standard error function (curve 6) suffers from overtraining after about 100 iterations—the error increases then starts to vary up and down. The resulting trajectory is shown in figure 6.10. The controller seems to have trouble damping the position p_2 after

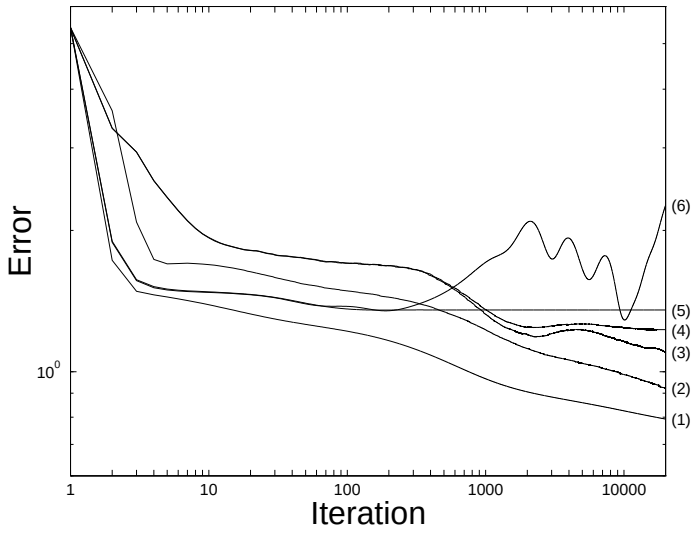


Figure 6.9: *Reference error vs number of iterations for different error functions. Key, curves from bottom to top: (1) Third order model, standard error,* $\alpha = 0.05$*. (2) Third order model, overshoot error,* $\alpha_1 = 0.1$*,* $\alpha_2 = 0.01$*. (3) Second order model, overshoot error,* $\alpha_1 = 0.1$*,* $\alpha_2 = 0.01$*. (4) Same as (3) but also with* $\beta = 0.00002$*. (5) Second order model, standard plus output limiting error,* $\alpha = 0.05$*,* $\beta = 0.001$*. (6) Second order model, standard error,* $\alpha = 0.05$*.*

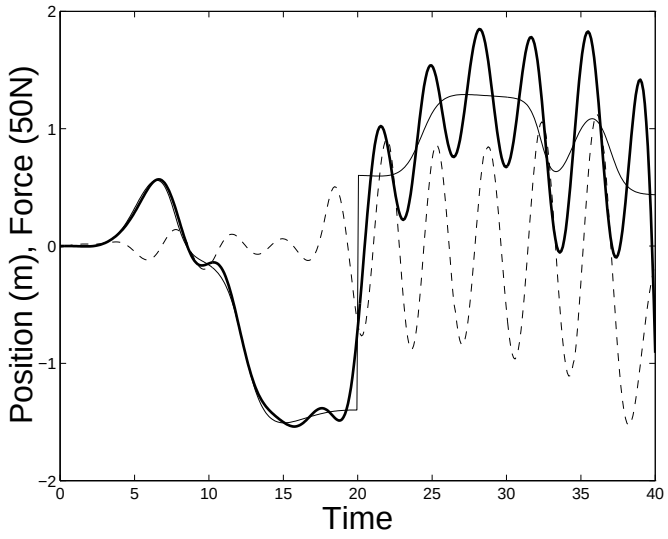


Figure 6.10: *The trajectory that results after training with a second order model and a standard error function. Key: reference (—), p_2 (**—**), $\mathbf{x}$ (– – –).*

the discontinuity occurs. This is because the second order model is an imperfect match for the crane's impulse response. Adding a small amount of output limiting ($\beta = 0.001$, curve 5) improves the situation—the error reaches its minimum value after about 100 iterations and no further training occurs. Output limiting prevents learning divergence, but it also prevents the error from going below a certain level.

Using the overshoot error function on the second order model provides better convergence (curve 3), although a small amount of over-training occurs. Adding some output limiting reduces the over-training (curve 4), although it also limits the minimum error that can be reached.

The third order model (curves 1 and 2) performs uniformly better than the second order model. The resulting trajectory for curve 1 is shown in figure 6.11. In this case the overshoot error function (curve 2) performs worse than the standard error function (curve 1)—in fact training takes longer but convergence is still achieved. This is because the third order model is a good match to the crane's impulse response.

In summary, more accurate models result in faster learning—less accurate models learn more slowly and can over train. Output limiting prevents over-training but limits the system performance. The overshoot error function seems to be a universally useful way to prevent over-training, and has better performance than output limiting. In a real system it might be preferable to perform output limiting even for an accurate model, to be consistent with the system's physical limitations (i.e. no real motor can generate an arbitrarily high force). But the above comments are only guidelines. In a real system a variety of different error functions should be tried, because performance is very dependent on the system dynamics and reference trajectories that are used.

6.4.4 Eligibility profile accuracy

How close should the eligibility profile model be to the system impulse response? As stated above, it is difficult to place formal limits on the eligibility model, but the following experiment will give an indication of the variation that is allowed.

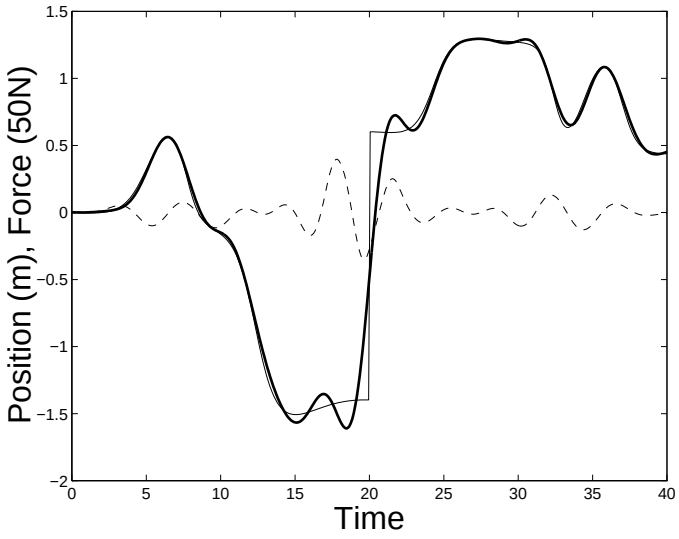


Figure 6.11: *The trajectory that results after training with a third order model and a standard error function. Key: reference (—), p_2 (▬), $\mathbf{x}$ (– – –).*

Figure 6.12 shows various third order eligibility profiles obtained by replacing the A matrix with A^k, where k (the "variation parameter") is in the range $0.5 \ldots 2$. This parameter controls the compression/expansion of the profile. Figure 6.13 shows the error that results after training when each of these profiles is used, with standard error ($\alpha = 0.05$) and overshoot error ($\alpha_1 = 0.1$, $\alpha_2 = 0.01$) training, and a discontinuous reference trajectory. Figure 6.14 shows the error performance of each profile for standard error training (the corresponding graph for overshoot training is similar). Standard error training is successful with k in the range $0.7 \ldots 1.8$. Overshoot error training is successful with k in the range $0.8 \ldots 2.0$. Other values of k cause the error to diverge. Note that higher k tends to result in lower error because the extra area under the eligibility profile curve effectively increases the learning rate. These results show again that a large amount of eligibility profile error is allowable.

6.4.5 History buffer size selection

How small can the history buffer size be without affecting training? This question is also related to the bigger question of how much eligibility profile mismatch is allowed, because a small history size truncates the eligibility profile in time. Figure 6.15 shows the error that results after training (for a third order profile) when history sizes from $50 \ldots 350$ are used, with the discontinuous reference trajectory. Any history size above about 200 produces the same training result, because after 200 time steps the eligibility profile has almost fully decayed (figure 6.3). A history size less than about 100 causes divergence as the truncated eligibility profile is too inaccurate. The conclusion continues to be that a large amount of eligibility profile error is allowable.

6.4.6 CMAC input configuration

Thus far all experiments have used a CMAC with the current time t as the only input. This has the advantage that FOX can anticipate an arbitrary reference signal as long as the reference value is keyed to

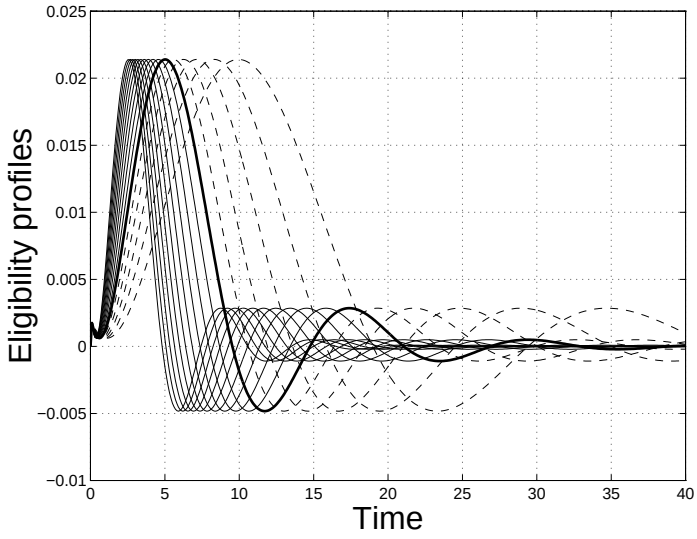


Figure 6.12: *Various third order eligibility profiles obtained by replacing the A matrix with A^k, where k (the "variation parameter") is in the range $0.5 \ldots 2$. Key: $k > 1$ (—), $k = 1$ (**—**), $k < 1$ (– – –).*

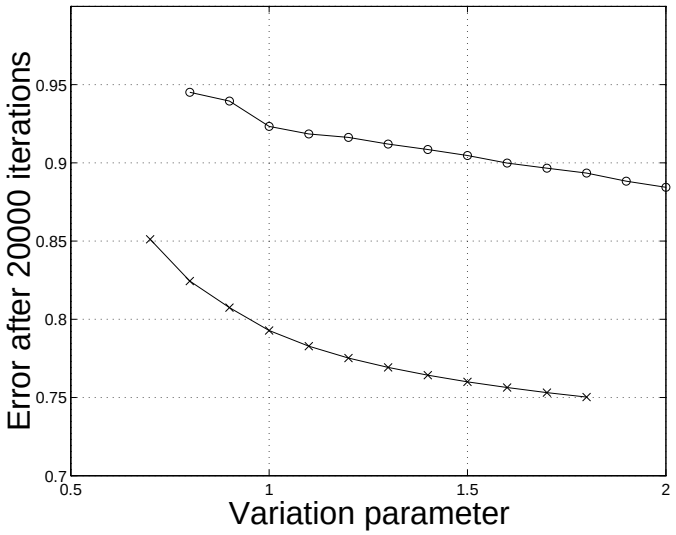


Figure 6.13: *Error versus variation parameter after 20000 iterations (third order eligibility profile). Key: standard error function ($\times$), overshoot error function ($\circ$). Unplotted points correspond to simulations with diverging error.*

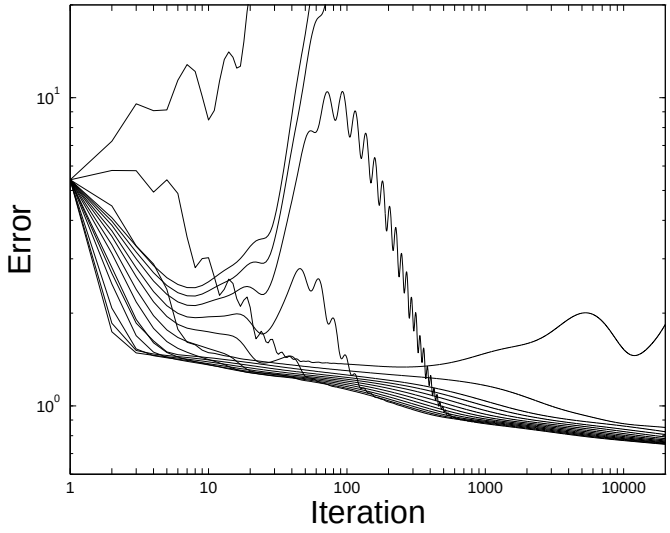


Figure 6.14: *Error performance with the various eligibility profiles shown in figure 6.12. The standard error function is used ($\alpha = 0.05$).*

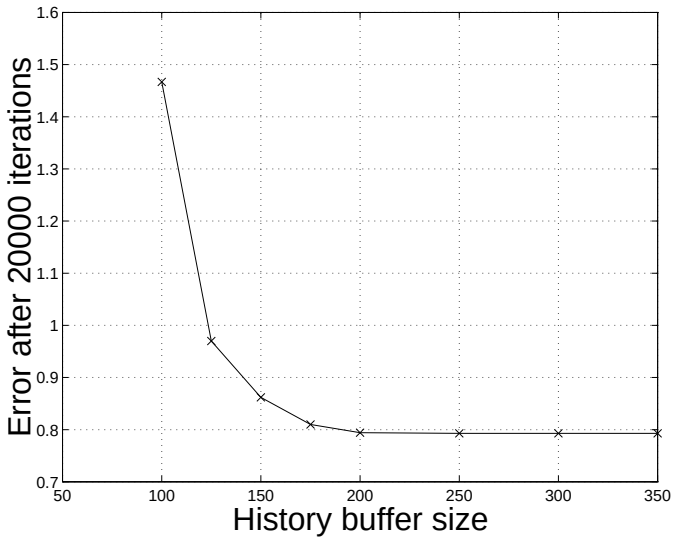


Figure 6.15: *Error versus history size after 20000 iterations (third order eligibility profile). A history less than about 100 results in divergence.*

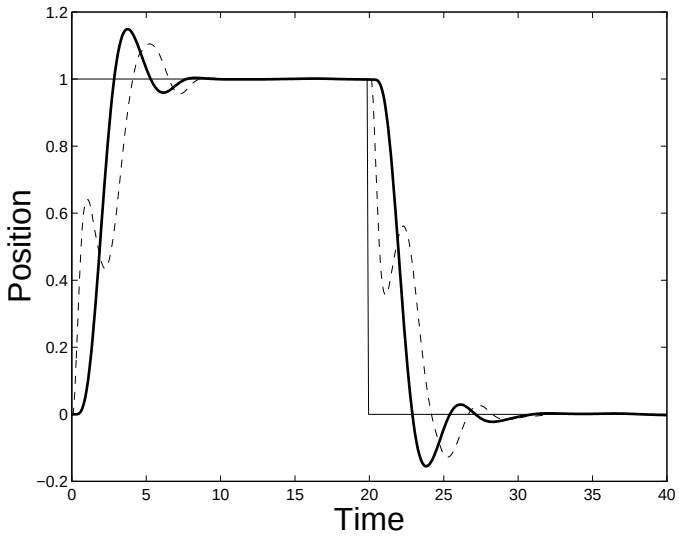


Figure 6.16: *The trajectory that results after training, with system state CMAC inputs and reference transition resets (third order eligibility model, standard error). Key: reference (—), p_1 (– – –), p_2 (▬).*

the time. However this has several disadvantages (outlined above) so in many situations it is desirable to use other input configurations.

A common alternative is to use the system state $\mathbf{y}$ as the CMAC input. Figure 6.16 shows the trajectory that results after 20000 training iterations with a third order eligibility model, and a standard error $\alpha = 0.05$. In this case an arbitrary reference can not be used because the CMAC input does not contain enough information to uniquely identify the current time. A constant or piecewise-constant reference value is commonly used (a two-level reference is used in figure 6.16). Whenever the reference is changed the FOX eligibility information must be zeroed (i.e. the FOX must be reset) as training is effectively being restarted from a different state. As a result the FOX controller can not anticipate the reference change at all. The CMAC input must be reference-independent, i.e. it must contain the position error relative to the current reference value not the position itself. Figure 6.17 shows what happens if a reference transition reset is not performed. Proper convergence can not be achieved because training for times $20 \ldots 40$ interferes with training for times $0 \ldots 20$ even though the two time periods are unrelated (in terms of cause and effect). As a reference, figure 6.18 shows the same system with a time CMAC input.

An alternative hybrid approach is to add an "anticipation" signal to the CMAC state inputs. The anticipation signal somehow indicates that a change in the reference signal is coming. This is demonstrated in figure 6.19, where the anticipation signal goes from -1 to +1 in the 5 seconds before the reference change. In this case resets are not required (in fact they would be disruptive). This combines advantages of both the time-input and state-input approaches (e.g. anticipation is achieved), but it is not always a practical solution.

Figure 6.20 shows the error performance of these three techniques. Time-input is the best in terms of convergence rate, state-input is worst, and the anticipation-signal approach is intermediate between the two.

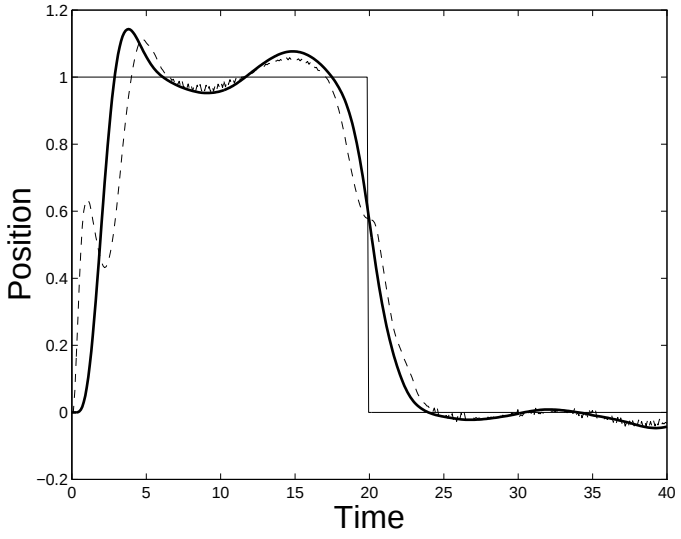


Figure 6.17: *The trajectory that results after training, with system state CMAC inputs and no reference transition reset (third order eligibility model, standard error). Key: reference (—), p_1 (– – –), p_2 (▬).*

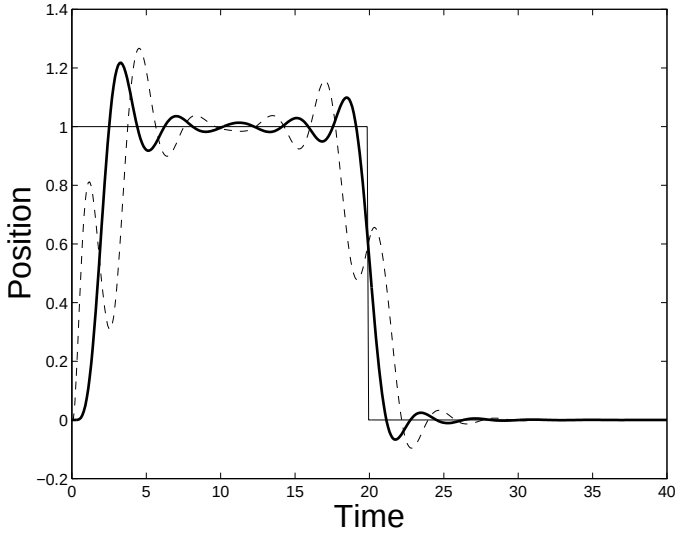


Figure 6.18: *The trajectory that results after training, with time CMAC input (third order eligibility model, standard error). Key: reference (—), p_1 (– – –), p_2 (▬).*

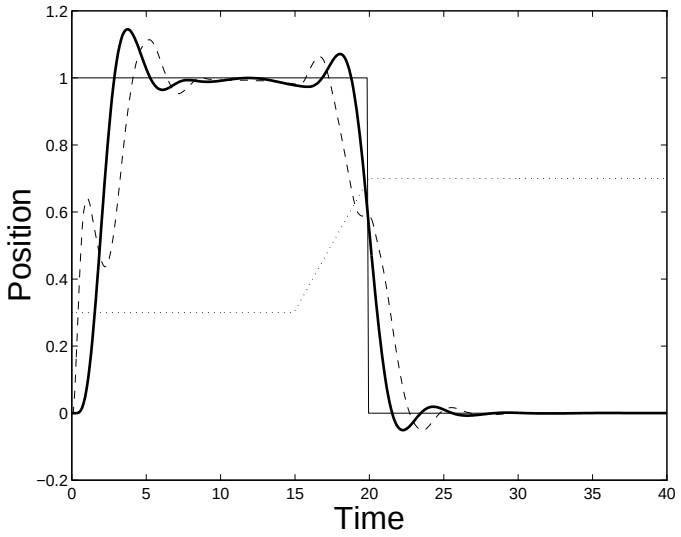


Figure 6.19: *The trajectory that results after training, with system state plus anticipation CMAC inputs (third order eligibility model, standard error). Key: reference (—), p_1 (– – –), p_2 (**—**), anticipation input ($\cdots$). Note that the anticipation input goes from $-1 \ldots 1$ but is scaled to fit the graph.*

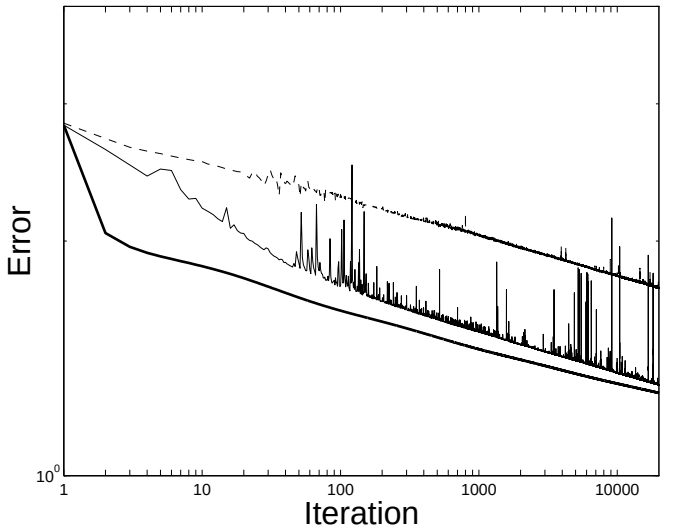


Figure 6.20: *Error performance with various CMAC input configurations (third order eligibility model, standard error). Key: time input (**—**), state input (– – –), state and anticipation input (—).*

6.4.7 Concluding notes

The following conclusions can be drawn from the crane simulation experiments:

- The eligibility model can not contain a mixture of large and small eigenvalues if a reasonable history size is required. Various techniques must be used to remove or replace bad eigenvalues.
- A large difference between the system's impulse response and the eligibility profile can be tolerated. Such differences arise from reduced order or inaccurate eligibility models, or short history buffers. The problem remains to quantify the exact amount of tolerable difference. Note that model accuracy does not necessarily equate with model order, although higher order models have the potential to be more accurate.
- Lower order eligibility models are good, because they consume less memory and computing power, and also because they are less likely to contain conflicting eigenvalues. However, models should not be simplified to the point where they are too inaccurate.
- More accurate eligibility models result in faster training. Less accurate eligibility models are susceptible to over-training. This can be corrected by using an output limiting or overshoot error function. The overshoot error function is generally useful—it makes many systems more stable and has a minimal cost in convergence rate.

6.5 Testing: Gantry crane

A FOX controller was constructed for the control of a model gantry crane. A picture of the experimental hardware is shown in figure 6.21, and a schematic is shown in figure 6.22. The crane can move horizontally on a set of rails. It is connected by a toothed belt and pulleys to a small DC motor via a gearbox (ratio 25:1). The crane supports a load (a lead weight) by a steel cable which is free to swing left and right.

An analog sensor measures the cable angle θ (in degrees) and a shaft encoder on the gearbox measures the crane position p_c (in meters). The load position p_ℓ is calculated from the cable length ℓ (assuming a small angle) by

$$p_\ell \;=\; p_c + \ell\,\frac{\pi}{180}\,\theta \qquad \text{(m)} \tag{6.22}$$

An electronics box contains the high power motor driver circuit which accepts a bipolar voltage from a D/A converter. There is also an A/D converter and shaft-encoder decoder circuit for sensor digitization.

A block diagram of the crane controller algorithm is shown in figure 6.23. Table 6.3 gives the experiment and CMAC parameters. A linear PD controller provides the "internal" feedback loop. It tries to keep the load angle at zero by implementing:

$$\mathbf{x}_\ell \;=\; -0.02\,\theta - 0.003\,\frac{\mathrm{d}\theta}{\mathrm{d}t} \tag{6.23}$$

These PD coefficients, which provide adequate control, were found by trial and error. The main feedback loop contains a FOX controller which is trained to make the load velocity equal to a desired velocity computed from the crane position error as follows:

$$\text{desired load velocity} \;=\; -0.6\left(\frac{1}{1+e^{-10(p_c-\text{ref})}} - 0.5\right) \qquad \text{(m/s)} \tag{6.24}$$

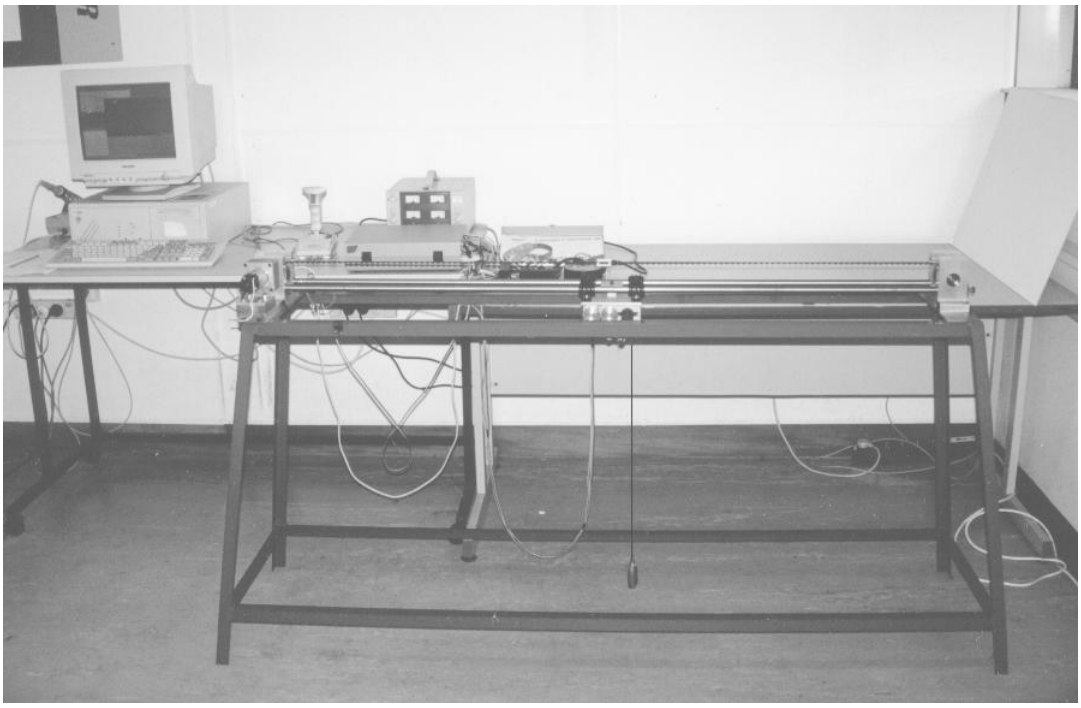


Figure 6.21: *A picture of the gantry crane experimental hardware.*

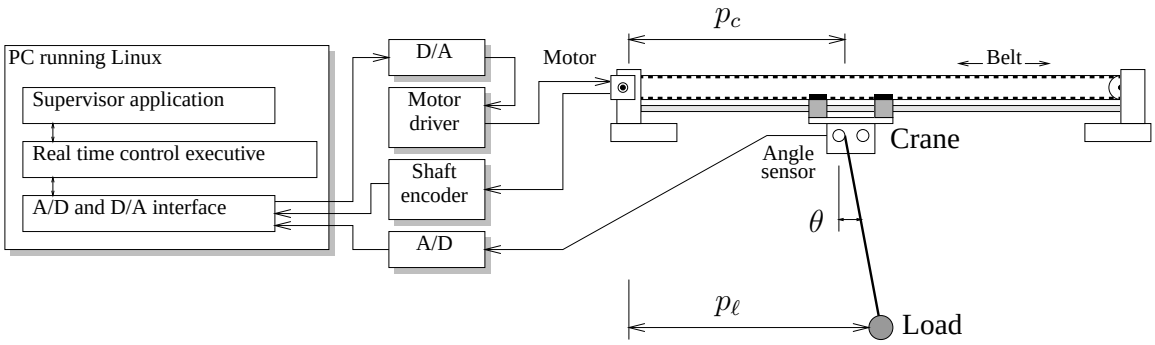


Figure 6.22: *A schematic of the gantry crane experimental system.*

System time step (h)	$1/64$ s
Standard error FOX learning rate (α)	0.0005
CMAC number of inputs	4
CMAC resolution on each input	200
CMAC number of weights	100000
CMAC n_a	20
FOX default eligibility model order	2

Table 6.3: *Standard gantry crane experiment parameters, including CMAC parameters. Most of these parameters are used for most of the experiments.*

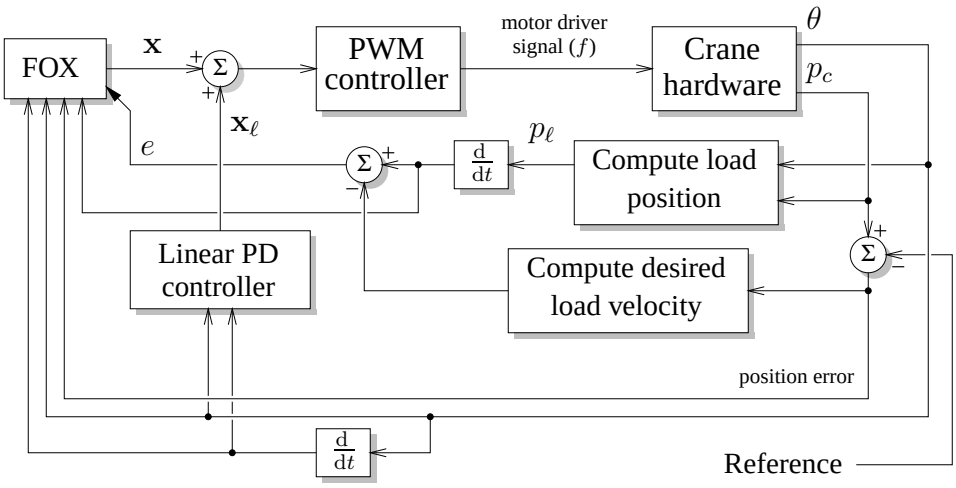


Figure 6.23: *A block diagram of the crane controller algorithm.*

Thus when the crane is far away from its reference position the desired velocity will be ± 0.3 m/s, and as it gets closer to the reference the desired velocity will smoothly ramp down to zero.

The time derivatives are computed from the sensor data using third and fifth order digital Butterworth IIR filters. A software PWM driver is used to try and compensate for stickiness within the gearbox, which prevents low motor drive signals from moving the crane. If the motor drive signal is above a threshold it is sent directly to the D/A converter. If it is below this threshold it is used to modulate a PWM signal whose duty cycle is calibrated to push the motor along at the appropriate speed. This provides a more linear low speed motor response, although it does cause the motor to emit an audible rattling sound at the system sampling frequency of 64Hz.

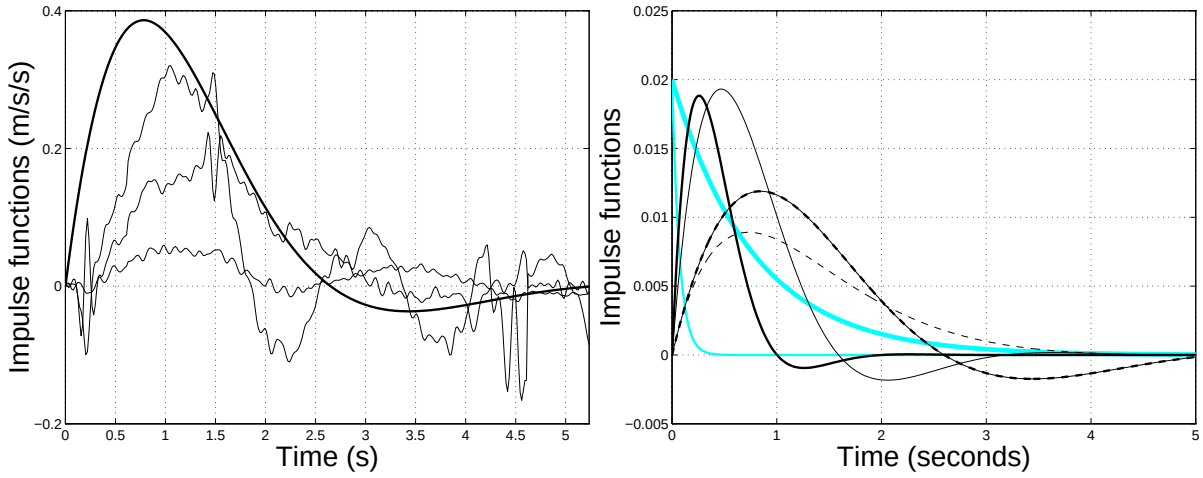
The crane mechanical system has four state variables[2]. The FOX inputs represent each state variable. Note that the position variables are reference independent. The reference signal is a squarewave that alternates between ± 3 every six seconds. The FOX is reset on each reference change.

6.5.1 Eligibility profiles

The gantry crane system is very nonlinear. Even an "ideal" crane model is intrinsically nonlinear, and this system also has nonlinear friction (mainly in the gearbox) and the drive electronics have a saturating response.

Strictly speaking, the impulse response of a nonlinear system is not a well defined concept—but the simplest approach is just to treat it as a linear system and try to measure the characteristic time constants of the system's step response. Figure 6.24a shows the error derivative for three different force steps. The responses have different forms because of system nonlinearities, though each one looks roughly like a decaying oscillation. The thick line in this plot shows a second order eligibility profile which roughly matches the experimental data. Its parameters were selected "by eye" (they are $k_a = 6$, $k_b = 3$, refer to Appendix E for the meaning of these constants). This profile was used for most of the following experiments.

[2] At *least* four—more accurate models will have more state variables.



(a) (b)

Figure 6.24: ***(a)*** *The three thin lines are measured impulse responses of the gantry crane system, equal to* $\mathrm{d}e/\mathrm{d}t$ *for a step force input. Each impulse was measured for a different force value. The impulses have different forms because of system nonlinearities. The thick line is the second order eligibility model used for most of the crane experiments.* ***(b)*** *Black lines: the four second order eligibility profiles used in figure 6.28a. Grey lines: the two first order eligibility profiles used in figure 6.28b. Key: second order* $k_a = 6$ *and* $k_b = 3$ *and* $\sigma = 200$ *(—), second order* $k_a = 18$ *and* $k_b = 6$ *and* $\sigma = 100$ *(—), second order* $k_a = 2$ *and* $k_b = 2.5$ *and* $\sigma = 250$ *(– – –), second order* $k_a = 2$ *and* $k_b = 1.5$ *and* $\sigma = 310$ *(– – –), first order* $k_a = 0.98$ *and* $\sigma = 200$ *(—gray), first order* $k_a = 0.8$ *and* $\sigma = 20$ *(—gray).*

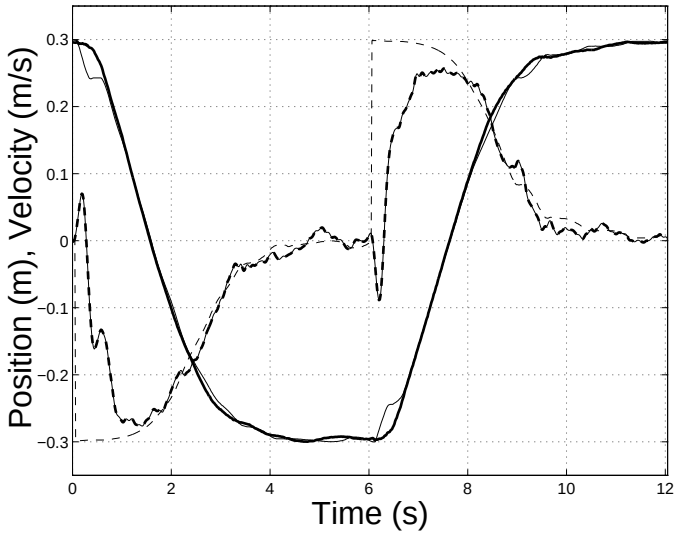


Figure 6.25: *Gantry crane experiment results. The best time domain performance achieved, with the overshoot error function,* $\alpha_1 = 0.0005$ *and* $\alpha_2 = 0.005$*. Key: cart position* x *(—), mass position* p *(—), desired* $\mathrm{d}p/\mathrm{d}t$ *(– – –), actual* $\mathrm{d}p/\mathrm{d}t$ *(– – –).*

6.5.2 Experiments

As before the following learning rate parameters were used: main rate (α), output limiting (β), output derivative limiting (γ), and the overshoot learning rates (α_1, α_2).

The best system performance was achieved using the overshoot error function with $\alpha_1 = 0.0005$ and $\alpha_2 = 0.005$. Figure 6.25 shows the time domain response that was achieved after 100 learning iterations, or 600 seconds (each iteration corresponds to one transition in the squarewave reference signal).

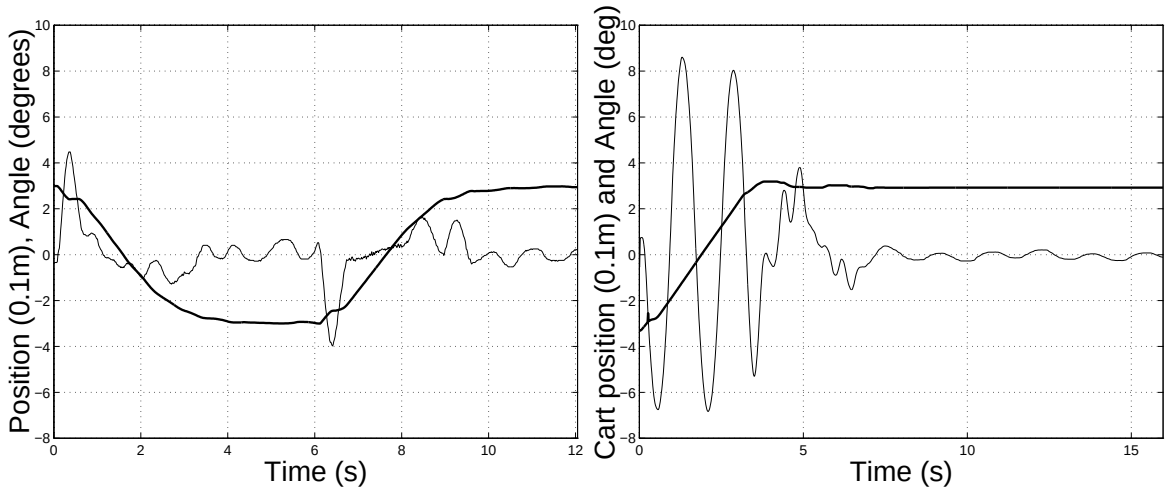
There are several points to note: the load velocity quickly reaches the desired velocity and then tracks it reasonably closely as the crane approaches its reference position. The crane has an initial quick movement to get the load up to speed. The short initial negative spike in load velocity is a quirk of the way load position is measured. The crane smoothly approaches and decelerates towards the reference. There is almost no reference overshoot. At all points the crane position and load angle are well controlled.

Figure 6.26 compares this performance with the best that was achieved using a linear PID controller [16]. The FOX based system achieves better control of the load angle and has less crane position overshoot.

Figure 6.27a shows the error performance for different overshoot learning rates. The total error for each six second iteration is computed as:

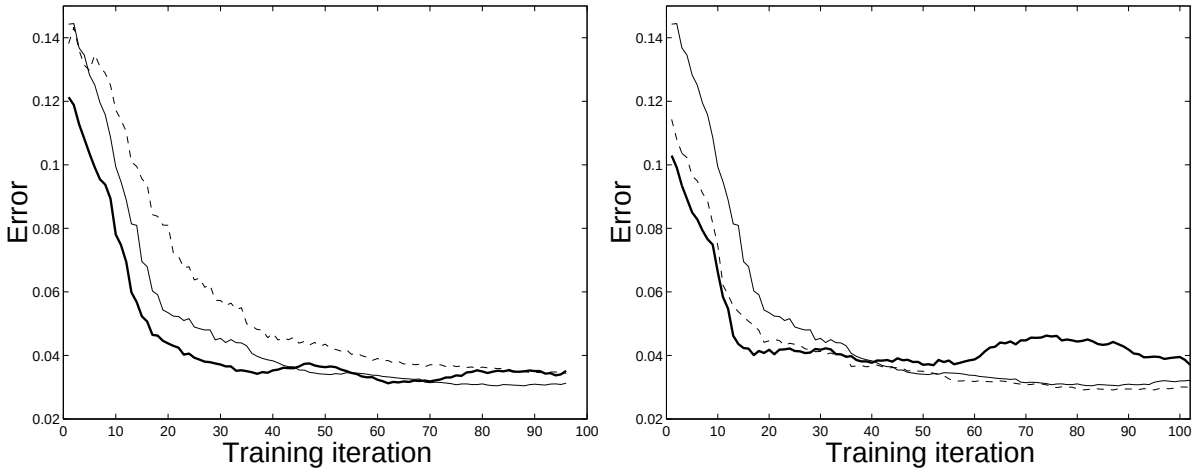
$$E = \sum_i h\, e_i^2 \tag{6.25}$$

This plot shows the expected behavior—increasing α_2 improves the training speed. Figure 6.27b shows the overshoot error performance when both α_1 and α_2 are varied. It shows that higher learning rates increase the training speed, but if they are too high ($\alpha_2 = 0.025$) then over-training will occur (the error eventually increases instead of monotonically decreasing). The time domain response in this case is shown in figure 6.30a. Notice how the crane position and load velocity oscillate instead of smoothly tracking their references—this is the hallmark of over-training.



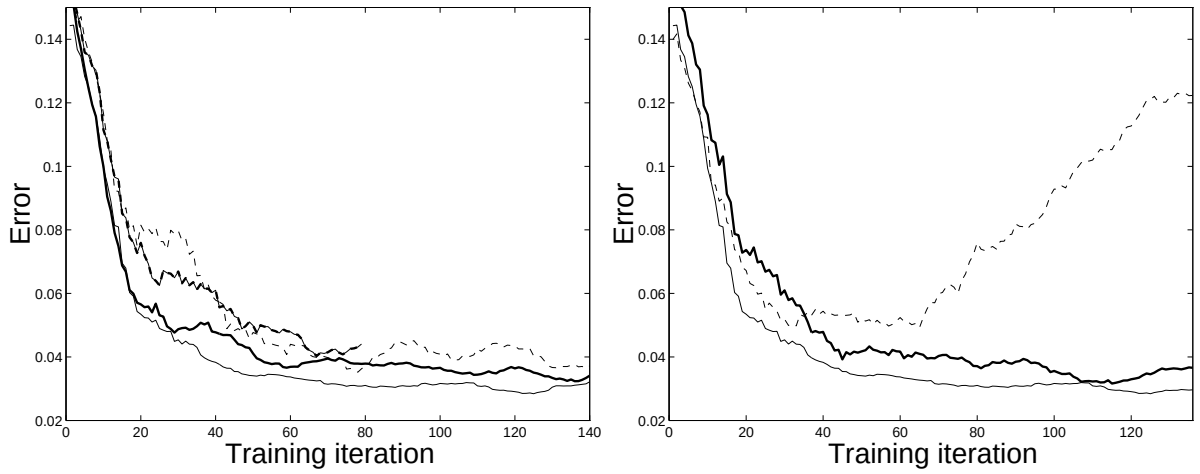
(a) (b)

Figure 6.26: *Gantry crane experiment results (time domain).* ***(a)*** *The same as figure 6.25 but only showing the cart position and load angle.* ***(b)*** *The best response achieved with a PID controller [16] (note that there is only one step change in the reference). Key: cart position x (—) load angle θ (—).*



(a) (b)

Figure 6.27: *Gantry crane experiment results.* ***(a)*** *Error performance for different overshoot error learning rates (all other constants have their standard values). Key: $\alpha_2 = 0.01$, $\alpha_1 = 0.0005$ (—), $\alpha_2 = 0.005$, $\alpha_1 = 0.0005$ (—), $\alpha_2 = 0.0025$, $\alpha_1 = 0.0005$ (– – –).* ***(b)*** *Error performance for different overshoot error learning rates (all other constants have their standard values). Key: $\alpha_2 = 0.025$, $\alpha_1 = 0.0025$ (—), $\alpha_2 = 0.005$, $\alpha_1 = 0.0005$ (—), $\alpha_2 = 0.001$, $\alpha_1 = 0.0001$ (– – –).*



(a) (b)

Figure 6.28: *Gantry crane experiment results.* ***(a)*** *Error performance for different second order eligibility profiles. The line styles correspond to the second order eligibility profiles shown in figure 6.24b.* ***(b)*** *Error performance for the first order eligibility profiles shown in figure 6.24b. Key: The best second order eligibility profile, as a reference (—),* $k_a = 0.98$ *(—),* $k_a = 0.8$ *(– – –).*

Figure 6.28a shows the error performance for different second order eligibility profiles, corresponding to the profiles shown in figure 6.24b. These profiles have a range of different decay constants but roughly the same form. In each case the learning rates were adjusted to achieve the best performance. They all result in roughly the same training behavior, although profiles that take longer to decay show slightly reduced performance.

Figure 6.28b shows the same thing for first order eligibility profiles. A first order profile with $k_a = 0.98$ results in worse performance than the best second order profile (although it is still acceptable). The profile with $k_a = 0.8$ is effectively zero-order, because it decays so quickly. It results in bad over-training after about the 60th iteration. The time domain response in this case is shown in figure 6.30b. The crane and load are subject to large oscillations, which only get worse with more training.

Figure 6.29 shows the error performance for the four different types of error function. In each case the learning rates were adjusted to achieve the best performance. The overshoot error function has the best performance, standard training (with no variety of output limiting) has the worst performance (with a significant over-training problem). Output limiting and output derivative limiting have intermediate performance, with little over-training.

6.5.3 Concluding notes

The following conclusions can be drawn from these experiments:

- FOX is capable of controlling nonlinear systems as well as linear ones.
- The overshoot-error training method provided the best performance, while standard-error training was the worst.
- Over-training occurred in all cases if the learning rate was too high.

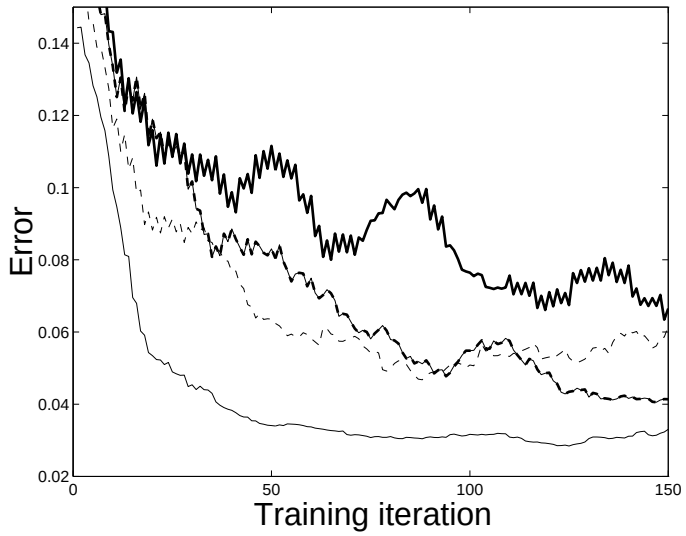
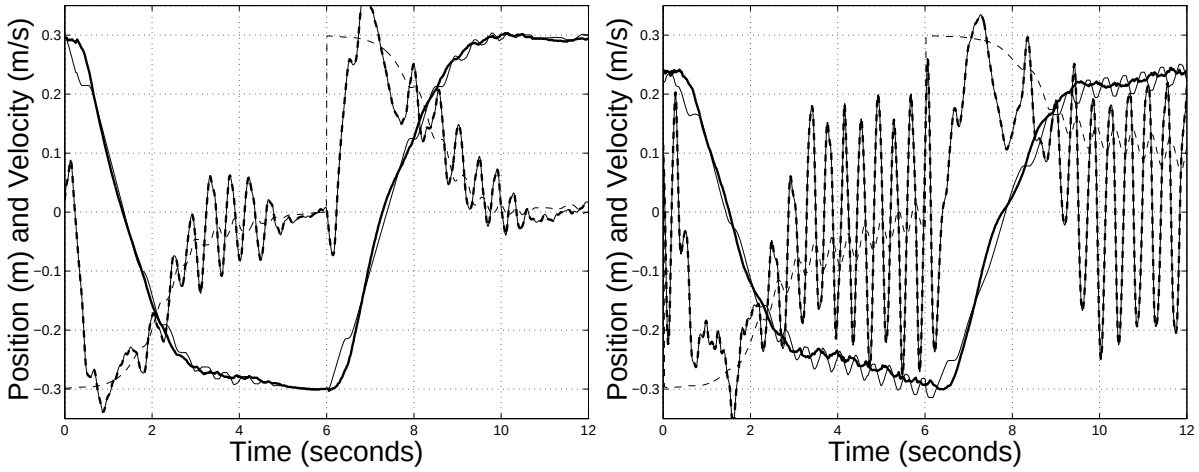


Figure 6.29: *Gantry crane experiment results. Error performance for different error measures. Key: overshoot training with $\alpha_2 = 0.005$ and $\alpha_1 = 0.0005$ (—), standard training with $\alpha = 0.0005$ (– – –), output limiting with $\alpha = 0.0005$ and $\beta = 0.0005$ (—), output derivative limiting with $\alpha = 0.0005$ and $\gamma = 0.0002$ (– – –).*



(a) (b)

Figure 6.30: *Gantry crane experiment results (time domain).* ***(a)*** *Response with the standard error function, $\alpha = 0.0025$.* ***(b)*** *Response with first order eligibility profile, $k_a = 0.8$. Key: cart position x (—), mass position p (—), desired $\mathrm{d}p/\mathrm{d}t$ (– – –), actual $\mathrm{d}p/\mathrm{d}t$ (– – –).*

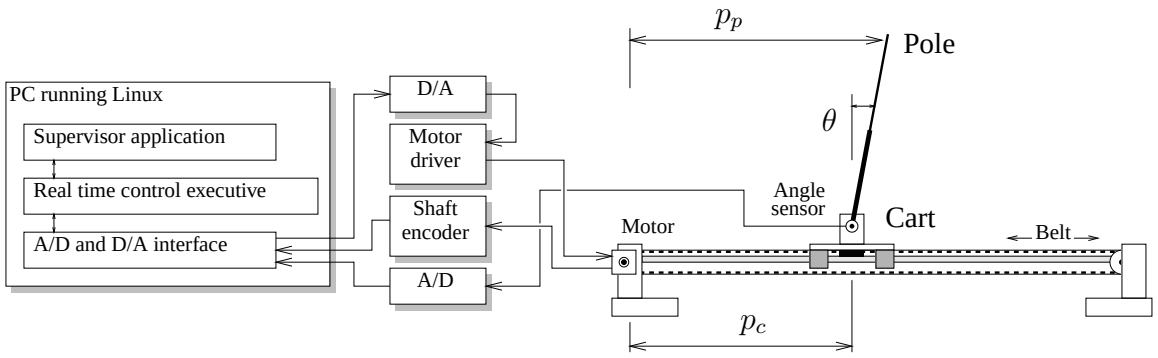


Figure 6.31: *A schematic of the inverted pendulum experimental system.*

- Large variations in the eligibility profile are allowed. Even a first order profile will work adequately for this system, but a zero-order profile will not. Thus eligibility is an important factor in the system's success—a straight FBE controller would not work.

6.6 Testing: Inverted pendulum

The fourth order inverted pendulum problem involves keeping a pole balanced on a "cart" by applying a horizontal force to the cart (figure 6.31). This is a widely studied "hard" problem in control theory, because although it can be solved adequately with standard non-adaptive linear control techniques, some extra elements of "intelligent control" are usually necessary for a good solution. Techniques using neural networks [47], reinforcement learning [10] and feedback-error with an expert rule base [42] have all been applied to it.

Two experiments were conducted to determine how easy it is for FOX to control this system, using the minimum amount of dynamical information. An AVI movie is provided on the CDROM that accompanies this thesis (`inverted.avi`) which shows most of the experimental results.

A schematic of the experimental system is given in figure 6.31 and two photos are shown in figure 6.32. The inverted pendulum system was based on the same hardware as the crane system, except that the crane platform was replaced with a cart-and-pole. A potentiometer sensor on the cart measured the pole angle θ, in degrees. As before, a shaft encoder measures the cart position p_c (in meters). The pole position p_p was calculated from the pole length ℓ (assuming a small angle) by

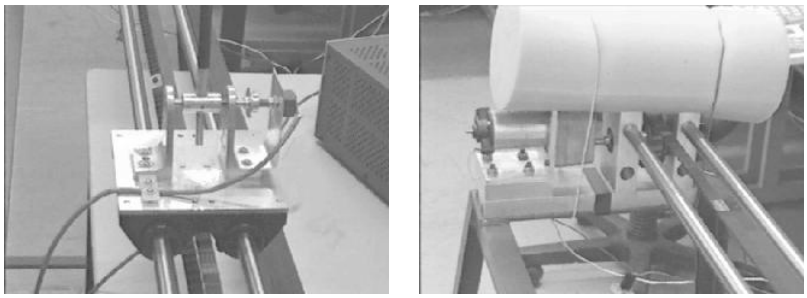
$$p_p \;=\; p_c + \ell\,\frac{\pi}{180}\,\theta \qquad \text{(m)} \tag{6.26}$$

The 25:1 gearbox was changed to a 5:1 ratio to get faster cart movement.

A block diagram of the inverted pendulum controller algorithm is shown in figure 6.33. Table 6.4 gives the experiment and CMAC parameters. A linear PD controller provides the internal feedback loop. It tries to keep the pole angle at zero and the cart position at a reference by implementing:

$$\mathbf{x}_\ell \;=\; -0.5(p_c - \text{ref}) - 0.25\,\frac{\mathrm{d}p_c}{\mathrm{d}t} - 0.06\,\theta - 0.000625\,\frac{\mathrm{d}\theta}{\mathrm{d}t} \tag{6.27}$$

The main feedback loop contains a FOX controller which is trained to get the cart position equal to the reference with the error $e = p_c - \text{ref}$. The desired pole angle is unspecified, but this is not a problem



(a) (b)

Figure 6.32: *Two pictures of the inverted pendulum experimental system. (a) The cart on its rails, showing the base of the pole and the angle sensor. (b) The motor, gearbox and drive-train.*

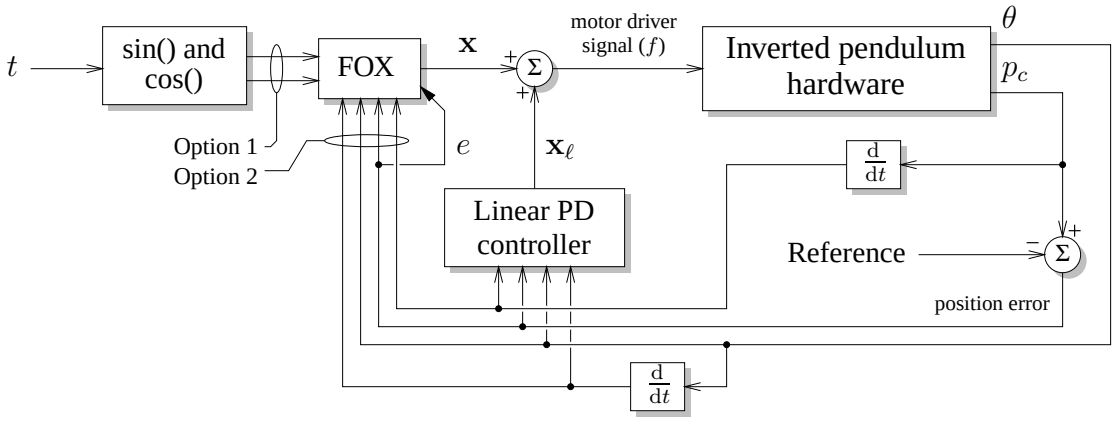


Figure 6.33: *A block diagram of the inverted pendulum controller algorithm.*

because the cart can not attain a reference position without the pole angle being simultaneously well controlled. Hardly any effort was made to choose good linear controller coefficients, on the assumption that the FOX controller would be able to compensate. Different CMAC input configurations were used for different experiments—they are described below.

6.6.1 Finding the eligibility profile

The impulse response of the inverted pendulum (with its linear controller) was measured by moving the cart to $p_c = 0$, carefully balancing the pole at $\theta = 0$, then gently tapping the top of the pole. As shown in figure 6.34, the cart quickly moved to compensate for the changing pole angle, but over-corrected and started to oscillate. The system converged to a limit cycle (due to its nonlinearity). No effort was made to tune the linear controller to reduce or eliminate this oscillation. Note that the flat spots in the cart position at its extremities are due to mechanical slip in the drive-train.

FOX can not model such an impulse response. And even if it could, it is unclear whether this is desirable because of the nonlinear nature of the system. Instead, a second order eligibility profile was chosen that roughly matches the time scale of the oscillations (figure 6.34). This eligibility profile decays

System time step (h)	$1/64$ s
CMAC number of inputs	2 or 4
CMAC resolution on each input	256 or 100
CMAC number of weights	100000
CMAC n_a	20
FOX second order eligibility model parameters	$k_a = 1.5$, $k_b = 1.7$, $\sigma = 384$

Table 6.4: *Standard inverted pendulum experiment parameters, including CMAC parameters.*

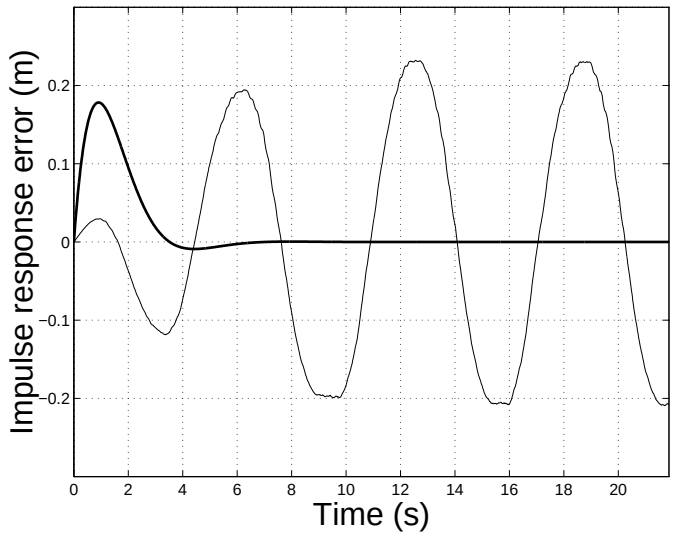


Figure 6.34: *Key: Impulse response (error e) of the inverted pendulum system (—). Eligibility profile model (**—**).*

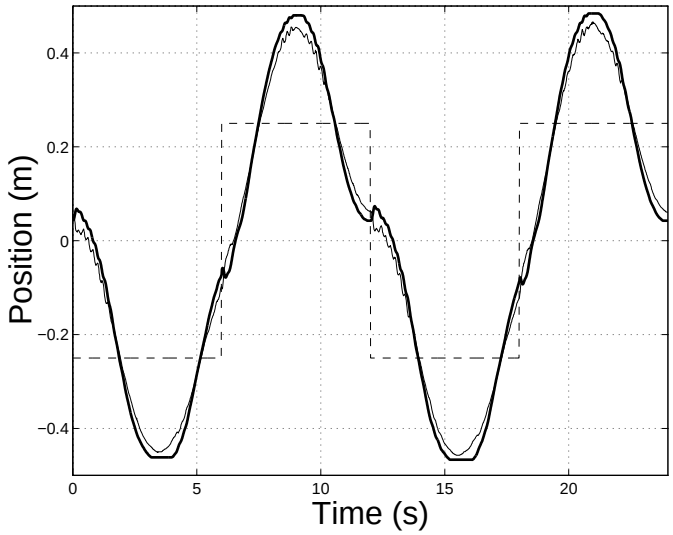


Figure 6.35: *Response of the feedback FOX inverted pendulum system to a squarewave reference, at the start of training. Key: cart position p_c (—), pole position p_p (—), reference (– – –).*

to zero, which is probably more appropriate for control than an oscillatory profile, as pointed out in section 6.2.1.

6.6.2 Learning feedback control experiment

In this experiment FOX was used in a feedback control mode, i.e. the CMAC inputs were the four system state variables (reference independent). A square wave reference was used that changed between ± 0.25 every 6 seconds. The FOX was reset on each reference change. It was determined that the standard error function with output derivative limiting gave a good response ($\alpha = 0.005$ and $\gamma = 0.003$).

Figure 6.35 shows the response at the start of training. The cart position occasionally crosses the reference, but it is clear that the linear controller is insufficient to the task. Figure 6.36 shows the response after 60 training iterations (360 seconds—each iteration is a transition of the reference). The response has improved greatly: the cart position smoothly approaches the reference and there is little overshoot. The pole angle is similarly well controlled. Convergence has been achieved at this point, as further training does not improve the response much (various electrical and mechanical nonlinearities mean that a perfect response is difficult to achieve). The output derivative limiting means that over-training is not a problem.

Of the 100000 CMAC weights, only 12359 were nonzero after training (12.4%). This implies that a much smaller weight set could have been used without affecting performance.

6.6.3 Learning feed-forward control experiment

In this experiment the task was to follow the reference signal

$$\text{ref} \quad = \quad \sin\left(\frac{\pi}{2}t\right) + \cos(\pi t) \tag{6.28}$$

which repeats every four seconds. In this case the CMAC inputs must somehow encode the reference position, so the FOX was used in a feed-forward control mode, with two inputs representing time t like

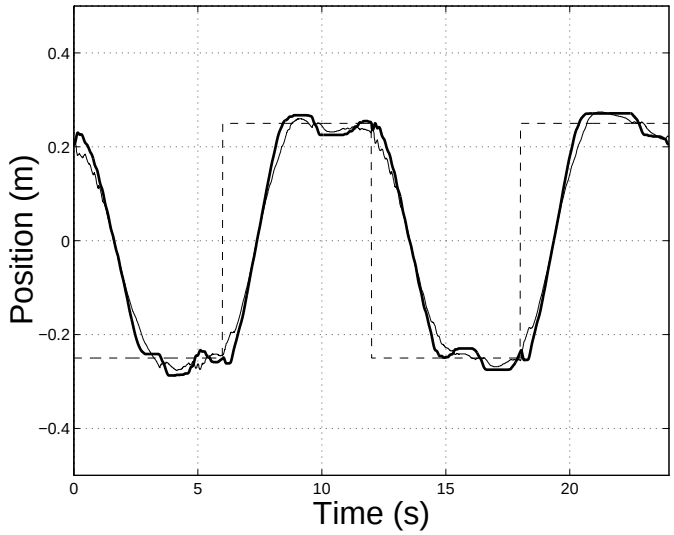


Figure 6.36: *Response of the feedback FOX inverted pendulum system to a squarewave reference, after 60 training iterations. Key: cart position p_c (—), pole position p_p (—), reference (– – –).*

this:

$$\text{input 1} \quad = \quad \sin\left(\frac{\pi}{2}t\right) \tag{6.29}$$

$$\text{input 2} \quad = \quad \cos\left(\frac{\pi}{2}t\right) \tag{6.30}$$

The time t by itself can not be used because then the CMAC would not generalize over those parts of the reference that are the same. The following "wrapped time" is also inappropriate:

$$\text{input 1} \quad = \quad \frac{t}{4} - \left\lfloor \frac{t}{4} \right\rfloor \qquad (\text{in the range } 0 \ldots 1) \tag{6.31}$$

because the input will jump discontinuously from 1 to 0 at the end of each cycle which will interfere with the CMAC's generalization ability. The CMAC input contains no information about the actual state of the system, so whenever the trained controller is used the system must start close to a known state or the behavior will be unpredictable. The standard error function was used, with $\alpha = 0.01$.

Figure 6.37 shows the response at the start of training. The cart is unable to follow the reference at all. Figure 6.38 shows the response after 120 cycles of training (480 seconds). The cart position now matches the reference closely. The pole position was unspecified by the error function, but training has given it a trajectory appropriate to the cart position. The error performance during training is shown in figure 6.39.

Of the 100000 CMAC weights, only 3762 were nonzero after training (3.7%). The weight usage is a lot smaller than for the first experiment because the range of input trajectories seen is far more restricted.

6.6.4 Concluding notes

Hougen *et al* [47] describe the inverted pendulum problem as so difficult that it has not been adequately solved by a learning controller until recently. However, their version of the problem gives the controller

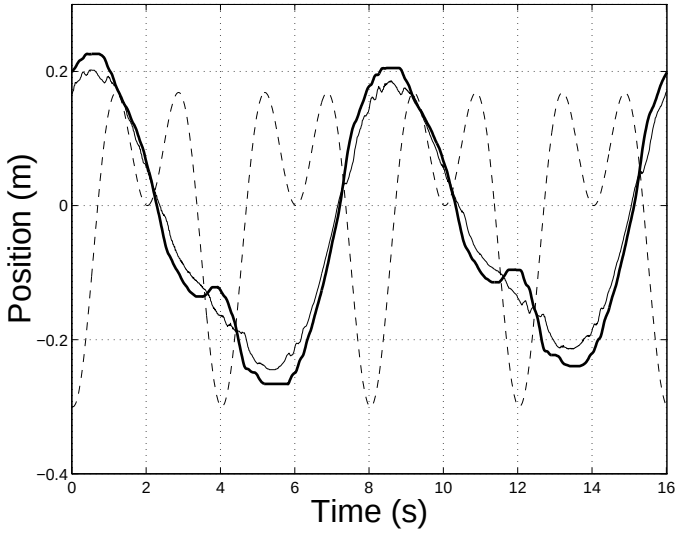


Figure 6.37: *Response of the feed-forward FOX inverted pendulum system to a non-constant reference, at the start of training. Key: cart position p_c (—), pole position p_p (—), reference (– – –).*

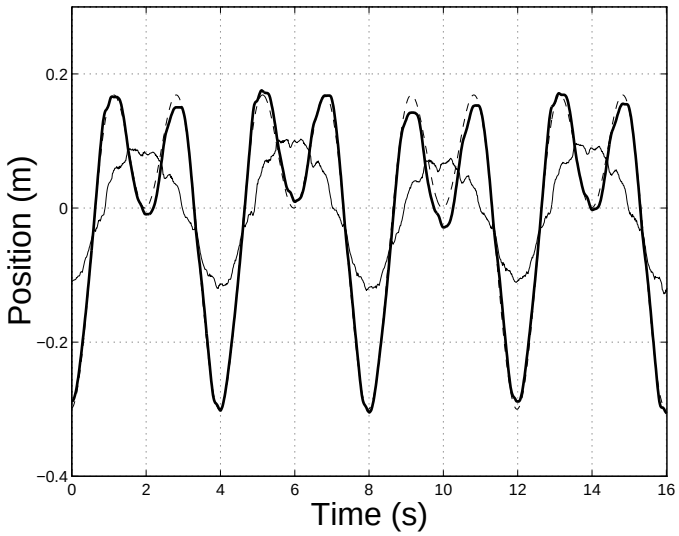


Figure 6.38: *Response of the feed-forward FOX inverted pendulum system to a non-constant reference, after 120 training cycles. Key: cart position p_c (—), pole position p_p (—), reference (– – –).*

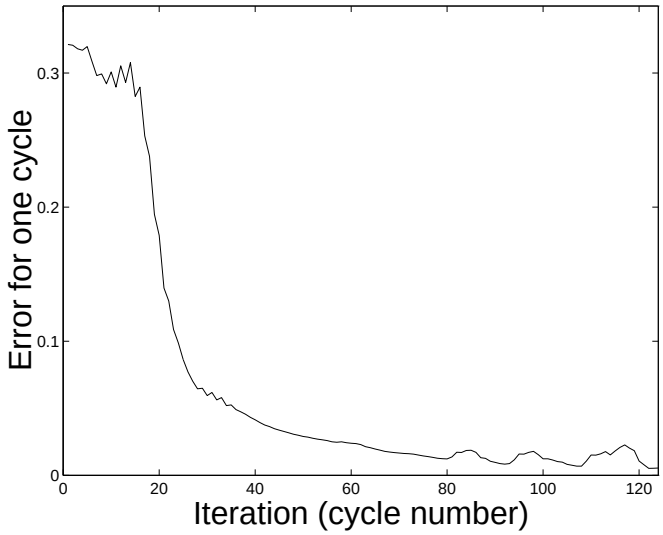


Figure 6.39: *Error performance during training for the feed-forward FOX inverted pendulum system.*

much less information about the system being controlled (only a binary reward/penalty signal is provided). In contrast the FOX controller has expert control knowledge split between the internal feedback controller and the continuous error signal.

The inverted pendulum controller was very easy to design. The most time consuming part was selecting an appropriate error function and learning rate by trial and error. Both experiments resulted in successful control being learned in a reasonable time. The control fidelity is very good considering the many mechanical deficiencies of the system. This system is competitive with the best currently available inverted pendulum learning controllers.

6.7 Testing: Mobile robot

A simple two wheeled mobile robot was constructed (see figure 6.40). The robot's task is to follow a thin guide cable laid out on the floor. The cable carries a 12kHz high-current signal. The robot has four pickup coils mounted in front of it to pick up the magnetic field from the cable. The position of the cable between each pair of sensors is inferred from the relative field strengths. The robot has two small DC motors for the left and right wheels. With the saturating motor driver electronics and sticky gearboxes these are two highly nonlinear actuators.

A schematic of the mobile robot experimental system is shown in figure 6.41. The 12kHz sensor signals are filtered by hardware on the robot to produce steady voltages that correspond to the field strengths present at each sensor, then they are digitized and passed to the software controller. The software controls two motor driver circuits which feed the left and right wheel motors.

A block diagram of the controller algorithm is shown in figure 6.42. The controller output is interpreted as a "turning" signal which is added and subtracted from a constant speed reference to get the motor driver signals. The raw sensor signals are processed by digital filters and two values are produced: an estimate of the cable position between the front pair of sensors, and similarly for the rear pair. The zero position for both sensors occurs when the cable is centered under the robot. The rear sensor position

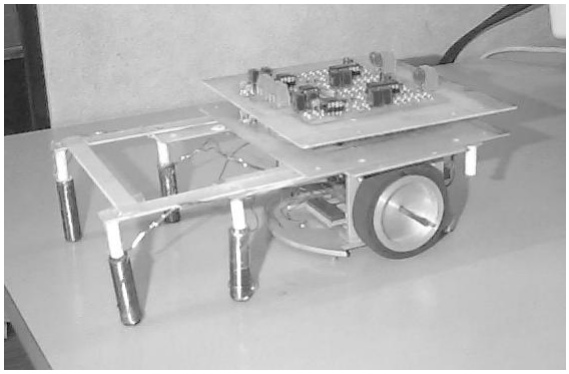


Figure 6.40: *The mobile robot. The four vertical poles mounted in front of the robot are sensor coils to pick up the magnetic field of the guide cable.*

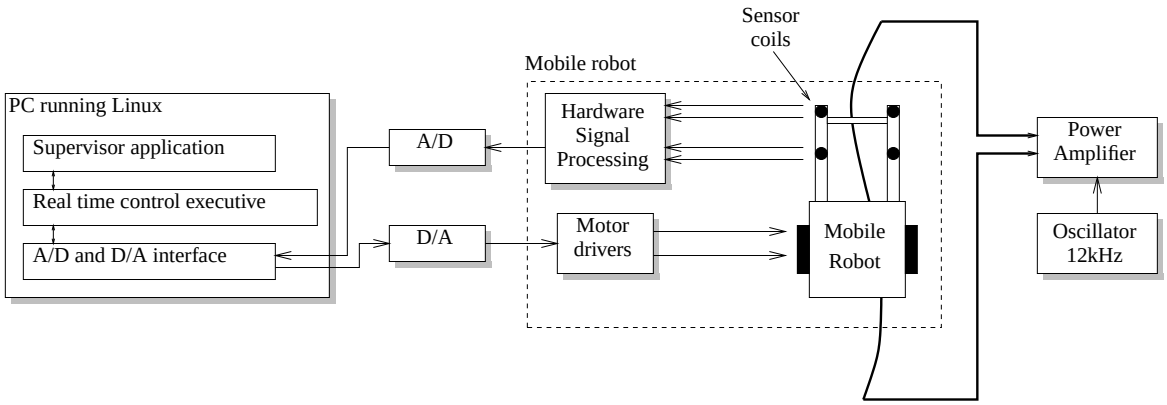


Figure 6.41: *A schematic of the mobile robot experimental system.*

is used as the error signal. When this is minimized the robot should be tracking the cable accurately.

The rear sensor position (and its derivative) are used in the internal linear feedback loop. The front sensor information is deliberately excluded, even though it can be usefully used in the linear controller. The reason for this is that FOX is expected to be able to *anticipate* the correct turning commands from the front sensor information, because the front sensor naturally sees bends in the cable before the rear sensor. The standard error function was used. The rest of the controller is configured similarly to previous examples.

With just the linear controller operating the robot has barely acceptable performance. Two typical robot trajectories are shown in figure 6.43. The robot can mostly keep itself on the track, but it overshoots every corner. About half the time it fails to take the sharp turn shown in figure 6.43b. Note that there is a magnetic anomaly present at the cable junction (where the two ends of the cable come together and lead off to the power amplifier) which causes brief sensor glitches.

The "impulse response" of the robot was measured in an extremely ad-hoc way: the robot was set to run up the center of a straight section of cable. Part way along it was nudged out of alignment and the error was measured as it sought the center again. The result is shown in figure 6.44a, along with the second order eligibility profile that was (arbitrarily) chosen to approximate it.

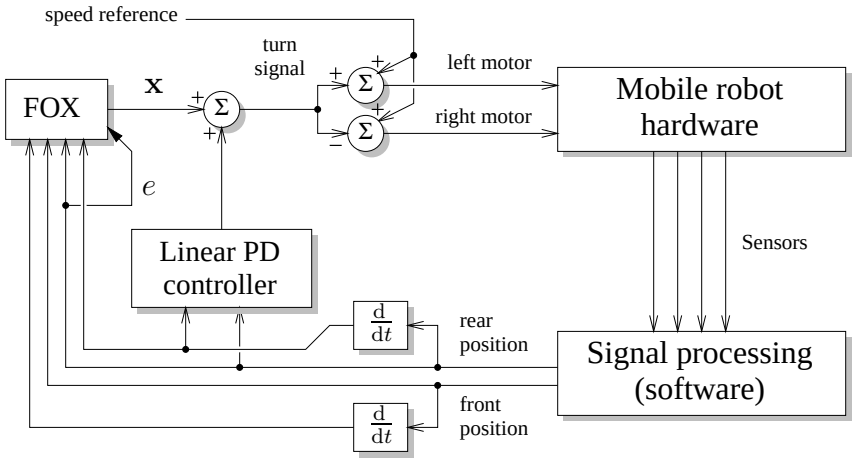
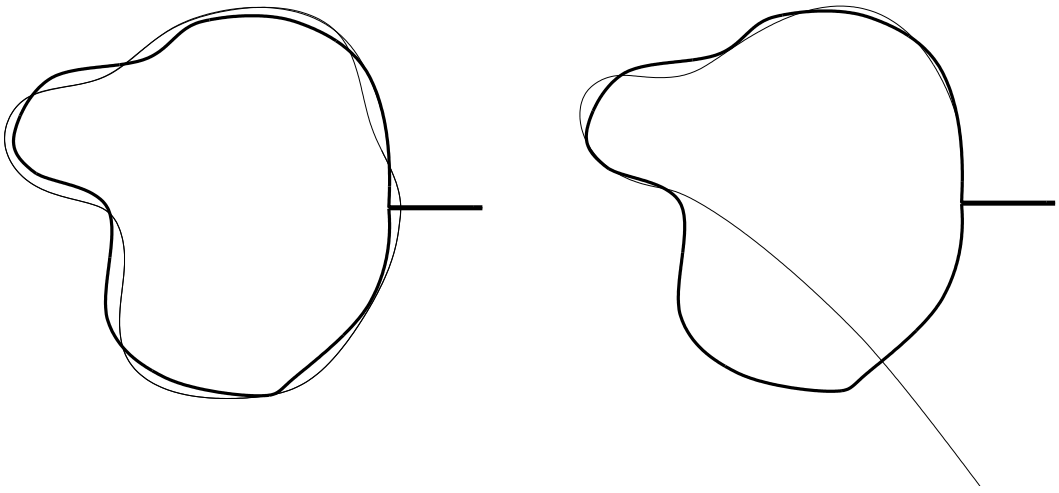
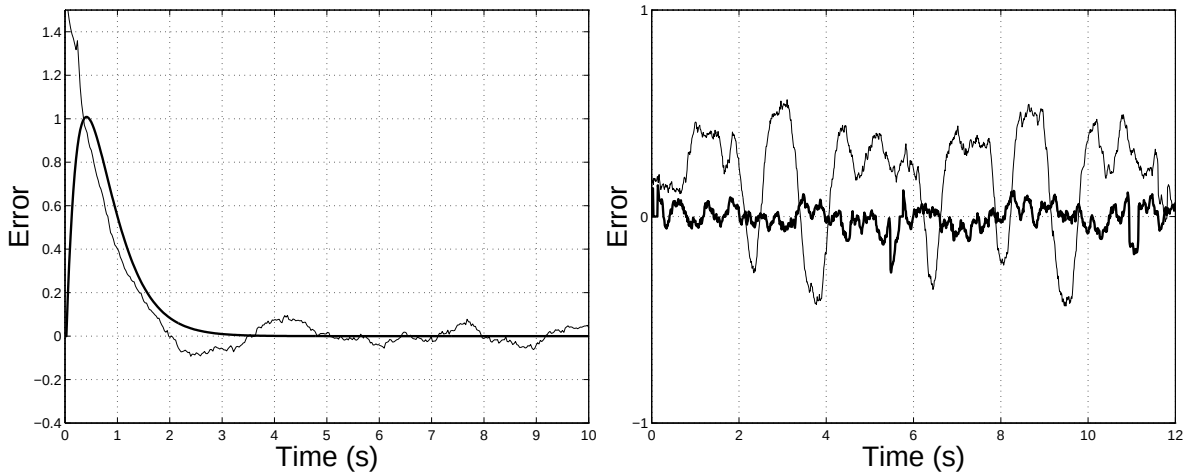


Figure 6.42: *A block diagram of the mobile robot controller algorithm.*



(a) (b)

Figure 6.43: *Mobile robot experiment: Two typical robot trajectories before training. The thick line is the position of the guide cable, the stalk at the right is where the two ends of the cable come together and lead off to the power amplifier. The thin line is the position of the robot (partially inferred from error data, and smoothed). The robot starts at the cable junction and travels anti-clockwise around the cable. In (a) the robot manages to stay on the track, in (b) it gets confused at a sharp turn and leaves the track. The robot is traveling anti-clockwise in both cases.*



(a) (b)

Figure 6.44: *Mobile robot experiment.* ***(a)*** *The (very) approximate impulse response of the mobile robot (—), and the eligibility profile chosen to match it (—), with $k_a = 0.44$ and $k_b = 1.33$.* ***(b)*** *The error trajectory before (—) and after (—) ten minutes of training using FOX.*

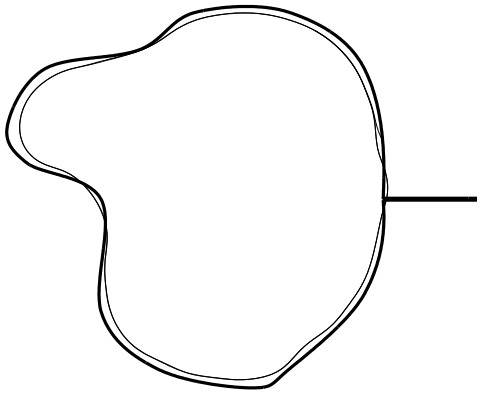


Figure 6.45: *Mobile robot experiment: The robot trajectory after training. The robot starts at the cable junction and travels anti-clockwise around the cable. Key: guide cable (—), robot trajectory (—).*

Figure 6.44b shows the results that are achieved after ten minutes of training with FOX. The error signal was greatly reduced in magnitude. Note that when the robot left the track during training the system was stopped and the robot was repositioned before continuing. This was done to prevent anomalous data from interfering with training. Figure 6.45 shows the robot trajectory after training. The track overshoots are reduced, and the robot never leaves the track. FOX successfully learnt to anticipate turning commands from the front sensor values.

FOX can only successfully operate in those areas of the workspace in which it was trained. This was illustrated by making the robot travel in a clockwise direction after it had been training in an anti-clockwise direction. The robot performance was instantly degraded, especially at the sharpest turn in the track.

6.8 Conclusions

It has been shown that FOX-based controllers are *easy* to design. Some control-systems experience is necessary to design the internal feedback controller (IFC), but this is a relatively easy task because the IFC only needs "acceptable" performance, so little time is taken up with parameter tuning. Once this is done, FOX requires only easy-to-measure dynamical information about the resulting system, i.e. the error impulse response. An eligibility profile model must be created from the impulse response, but a large amount of approximation is acceptable. Thus the eligibility profile can be designed "by eye" in many cases, so lower order models can be used (which are computationally smaller and faster).

The concept of CMAC weight eligibility was important for the success of the experiments described here. In all cases a zero-order FOX, equivalent to a feedback-error controller using a straight CMAC, did not work.

The selection of the CMAC inputs depends on whether the FOX is being used in a feedback or feedforward mode, and on the nature of the reference signal. The selection of the other CMAC parameters (for example n_a and n_w) is largely a matter of engineering experience, as each selection involves a variety of tradeoffs. As shown in the inverted pendulum example, the weight table occupancy can be used as a rough guide to the selection of n_w.

FOX is not just limited to linear systems. It can be applied to any nonlinear system where the concept

of the impulse response is sufficiently well defined. That is, the system's behavior when presented with an impulse should be sufficiently consistent over its useful state space, so that the eligibility profile always remains a reasonable approximation of the system's dynamics. Note that this depends on the fact that FOX can handle large eligibility profile / impulse response deviations. FOX will even work with non-decaying impulse responses.

The principle design decision is the selection of the error function. Different functions result in different system behavior, which is not always easy to predict. The most time consuming aspect of the design is trying various error functions and learning rates to see which is most effective. The learning rates must be selected to prevent over-training, while at the same time providing a quick enough convergence rate. The overshoot error function appears to be the most universally useful.

Chapter 7

Autonomous Robot Motion Control

7.1 Introduction

It was seen in the previous chapter that FOX can successfully learn to control a variety of mechanical systems. These systems were simple, in the sense that their desired behavior was easily specified (e.g. keep the inverted pendulum upright, or keep the robot on the track). Thus FOX could control the system by optimizing a single error value. This is adaptive control, but it is not *intelligent* control.

This chapter explores the more difficult problem of learning to control an autonomous robot that has to survive in a complex environment, and whose desired behavior is only known in vague terms. Again, note that only the motion control aspect of this problem will be explored (control of high level behavior is a separate issue). Through trial and error a variety of design principles have been developed that are useful for designing such systems using FOX. These will be applied to controlling a hopping monoped robot and a walking biped robot. These robots are simulated, not real. The virtual environment that they inhabit will also be described. Several MPEG movies have been created that illustrate various aspects of robot motion and training. These movies are stored on the CDROM that accompanies this thesis. They will be referred to thus: movie `foo.mpg`.

7.2 Specifying behavior

An autonomous robot designer often only has a vague idea of the behavior that is required. For a wheeled robot the challenge is to intelligently control the high level aspects of behavior, for example specifying the direction and speed of the robot over time to achieve some task. Legged robots must use intelligent control at a much lower level, because the control problem is dynamically complex and it is not obvious what leg trajectories will be effective.

7.2.1 Hard-wired and generic controllers

Controllers for autonomous robots can usually be classified between two extremes: hard-wired and generic. A hard-wired controller is simply one that has been constructed to implement the desired behavior (and no other). It has no adjustable parameters (that is, nothing to learn), but it is pre-adapted to its environment. Examples include Beer's artificial insect [15] and Brooks' hexapod [20].

A generic controller is at the other extreme: it contains neural networks or some other structures which are capable of implementing arbitrary control rules. The desired behavior must be specified using some error or reward signal, as the controller starts out by knowing nothing. It has many parameters, all of which must be learned as the robot interacts with its environment. Examples include Millan's TESEO system [81] and MLP controllers such as [102].

A practical system needs to compromise between these two extremes. A fully hard wired approach can not adapt to unanticipated variations in its environment. A fully generic approach may take too long to train (or may have to be trained offline, or may be incapable of learning adequate control rules, see below). A partially hard-wired controller with a range of adjustable parameters should be designed. Thus best use of the designer's expert knowledge about the problem domain is made by spreading it between the hard-wired controller design and the choice of error/reward signals. This is also usually more time efficient than either extreme: parameters can be learned that would otherwise have to be adjusted by the designer through many design iterations in the hard wired approach.

The number of parameters to learn is a design decision that depends on the problem: more free parameters makes the controller more flexible, but can also increase the difficulty of learning.

7.2.2 Generic controllers using scalar error signals

The desired behavior is commonly defined as that which minimizes a simple scalar error value. This is the approach for which the FOX controller has been formulated, and the one that was used in the previous chapter. Three common approaches for making use of scalar errors are gradient descent, genetic algorithms and reinforcement learning. These techniques are used more for generic controllers. Consider, for example, how a biped robot could be made to walk where the error signal just tried to maximize the robot's forward speed. This error signal does not give any indication of *how* the task is to be achieved, for example it does not constrain the legs to move in a stepping pattern. In general, for more complicated systems a scalar error provides less information about what each component of the system should be doing.

It is common for the error gradient vector (with respect to the system control parameters) to be known, in which case a gradient descent technique can be used (such as conjugate gradient, see [100]). However gradient descent may be unable to discover complex control actions in a completely generic controller, as it relies on the existence of a continuous (or analytic) search space, but most interesting problems will be discontinuous. Also gradient descent can easily become trapped in local minima and fail to find any acceptable behavior, especially with complex systems [49]. Generic-controller gradient descent techniques are most useful in simple systems.

Genetic algorithms [36] are a class of techniques that are capable of discovering unique controller structures and patterns of movement using simple error signals [111, 112]. They are extremely flexible and they do not become trapped in local minima, but they do not easily lend themselves to on-line learning in real robots (as large populations of automatons are required). Also they can be extremely slow.

Reinforcement learning systems such as temporal difference [117] and *Q-learning* [69] can discover low error actions directly. These methods have a strong theoretical foundation. Typical implementations rely solely on scalar reinforcement signals from the controlled system to train a homogeneous neural network. Although in principle they can learn control strategies which minimize penalty, in practice their lack of behavioral constraints make them slow to converge and very computationally intensive to train.

7.2.3 Analytical motion

Many authors have described algorithms for generating limb trajectories that achieve some goal. For example, [75] describes pattern generating algorithms that produce obstacle avoiding limb trajectories, and [127] describes how recursive workspace multi-body dynamics techniques are used to control a planar biped. Such analytical methods require good models of the robot to be available (its exact geometry and mass distribution), and they can sometimes be very computationally intensive.

7.3 General design principles

Through much trial and error a number of principles have been found to be useful when designing autonomous robot controllers. First, a "toolbox" approach should be used—that is, don't insist on using just one paradigm (e.g. neural networks, reinforcement learning, or symbolic AI) because each approach has its advantages and disadvantages. Hybrid designs which use a variety of techniques can often be more robust.

The robot's behavioral repertoire can be divided amongst multiple interacting modules, possibly ordered in a hierarchy. The robot's behavior *emerges* from the interaction of these modules. This approach can allow the designer to be vague about the behavior required. The controller must learn low-penalty actions within the constraints of the behaviors defined by its internal structure.

A compromise is made between hard-wired and fully learned behavior. The designer's domain knowledge is used to design a controller that has the potential to adapt the robot well to its environment. Some parameters are left unspecified (maybe they are difficult to find beforehand, or perhaps they depend on the specifics of the robot's environment). These parameters are learned by FOX modules, which work independently (or together in cases where a single parameter has multiple constraints). The error signals for the FOX modules are selected independently, they are not derived from a global performance error. Thus the whole-body optimal behavior is not predefined, but nevertheless the behavior should improve as learning occurs. This approach will obviously work better if the parameters have relatively independent influences on the system behavior. This approach can not learn fundamentally new behavior, but it can improve existing behavior, which will be quite adequate in many situations.

Note that unlike MLPs, CMACs can not discover new internal representations (in other words they do not perform global generalization), and so the CMAC parameters must be chosen carefully beforehand.

Some more specific design techniques will be described later in this chapter.

7.4 The simulation and virtual environment

The hopping and walking robots were simulated and visualized using a suite of applications developed in C++ by the author. Full details are given in Appendix H, but briefly:

- `RoboDyn` is a C++ class library that simulates the dynamics (i.e. the motion) of articulated rigid bodies. The numerical engine is based on the work of Scott McMillan [76, 77, 78, 79]. RoboDyn provides very efficient simulation: it uses Featherstone's recursive articulated-body algorithm [34] which executes in $O(n)$ time where n is the number of links in the system. Further details are given in Appendix I.

- `Dyson` is a language and compiler for specifying arbitrary dynamical systems. It was used to implement the robot controllers. It has the following components:

Constant	Value
Gravitational constant	9.81 m/s^2
Ground planar spring constant	0 N/m
Ground normal spring constant	10000 N/m
Ground planar damper constant	250 Ns/m
Ground normal damper constant	250 Ns/m
Ground static friction coefficient	2.5
Ground kinetic friction coefficient	1.8

Table 7.1: *Some environmental constants for the biped robot simulation.*

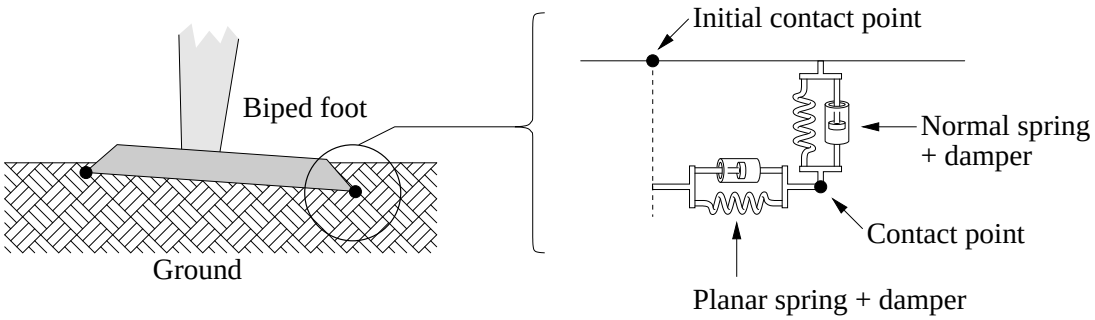


Figure 7.1: *Ground contact forces are modeled as two spring+damper units between the point of first contact and a collision detection point on the solid. Forces normal and planar to the ground are handled separately.*

  - A dynamic network system description language called 'DND'.
  - A compiler called '`dnd2exe`' (written in Perl) to convert DND into executable C code.
  - A simulation kernel (written in C) to simulate the system.

  See Appendix J for further details.

- `CyberSim` performs the actual robot simulation, it brings together `RoboDyn` with the output of `Dyson`. See Appendix H for further details.
- `CyberView` allows the user to move through a virtual 3D environment and observe the operation of the robot over time (the output of `CyberSim`). The custom graphics engine (the `R3D` library) supports flat shaded rendering to an X-terminal or texture mapped real-time rendering to a 3dfx/Voodoo graphics accelerator card. See Appendix H for further details.

In the virtual environment, the ground was modeled as a spring-plus-damper system with independent coefficients for normal and planar sliding contact (see figure 7.1). Some of the ground and environment parameters are given in table 7.1. Both viscous and Coulomb friction were modeled, and both static and sliding contact modes were supported, each with different coefficients. Every robot joint had built in spring-plus-damping limits on its motion, as well as internal viscous and Coulomb friction.

However, the mechanical models were unrealistic in several respects. Ideal torque-motors were used instead of more realistic (and dynamically complex) electromagnetic motor models. There was no slip-

page between the actuators (motors) and joints. The joint sensors were assumed to have unlimited resolution and be free from noise. And finally, no collision detection was performed between the robot links. This is most apparent in the biped experiments, where the biped's feet occasionally move through each other.

Hence despite the sophistication of the simulation, it is definitely not as good as a having a real robot when it comes to proving a new control methodology. However, real robots were not constructed for two reasons. First, they are obviously expensive and difficult to build. Second, they hinder a trial-and-error approach to controller construction and learning, because the mundane details of performing experiments on them are extremely time consuming when compared to simulation. This is not an argument for abandoning hardware, just a statement that simulation is particularly useful when new and relatively untried methods are being tested.

7.5 Hopping robot

7.5.1 Introduction

A simulated single legged hopping robot was created to provide a simple test of the above controller design strategy. Figure 7.2 shows the hopping robot in its virtual environment. The robot is roughly one meter high and has a 10kg body supported on a telescopic leg. An actuator can apply a force to the leg to make it extend or contract. "Hip" actuators are able to rotate the leg around a ball and socket joint that attaches it to the body. The robot's sensors measure leg and hip position, foot contact with the ground, the body speed along the x and y axes (parallel to the ground), and the slope of the ground under the foot (along the x and y directions). The robot's task is to follow an approximately square trajectory along the ground, which takes it up and down a ramp.

7.5.2 The basic controller

An outline of the robot's controller is shown in figure 7.3. It is similar in principle to the hopping robot controllers described by Raibert [103] and Boone [17]. The control problem is decoupled in to three relatively independent problems: control of jumping height, control of body angle while the foot in on the ground, and control of leg position when the foot strikes the ground (to indirectly achieve speed control).

All actuators are controlled using 'LLJC' (low level joint controller) modules which implement simple proportional plus derivative control:

$$\text{force} \quad = \quad \text{gain} \cdot (\text{ref} - \text{pos}) - 100 \cdot \frac{\mathrm{d}}{\mathrm{d}t}(\text{pos}) \tag{7.1}$$

Where 'pos' means position and 'ref' means reference. For the hopping robot the gain is 1000 unless otherwise specified. A simple state machine selects the gain of the LLJC that controls the leg length, to achieve a hopping motion (state 1 selects the high gain which thrusts the leg out).

There are two hip actuators to control the x and y rotation of the leg to allow hopping in any direction. For simplicity, only the controller for one axis is shown in figure 7.3. When the foot is on the ground an LLJC controls the hip actuator to try and achieve a zero body angle ϕ_b. This would eventually cause the robot to tip over if the foot stayed on the ground. When the foot is off the ground another LLJC controls the leg angle relative to the body. The reference is determined from the following control rule, which tries to maintain the forward speed of the body at some reference value:

$$\text{ref} \quad = \quad k_1 \cdot \text{speed} - k2\ \phi_b - k_3\ (\text{desired\_speed} - \text{speed}) - k_4\ \text{slope} \tag{7.2}$$

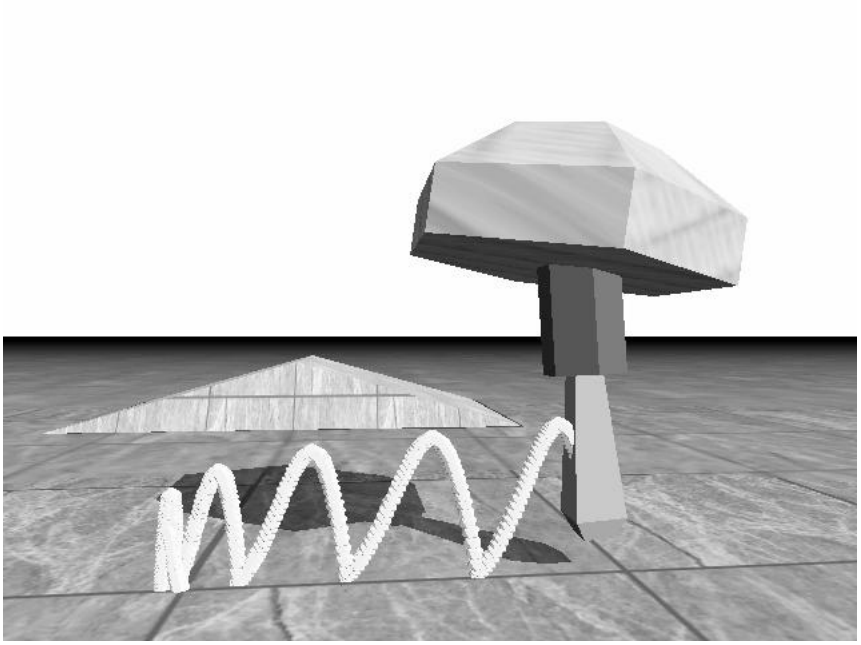


Figure 7.2: *The monoped hopping robot in its virtual environment. As the foot moves it leaves a trail behind, which is useful for showing the robot path in these still images. In the background is a ramp which the robot must climb and descend.*

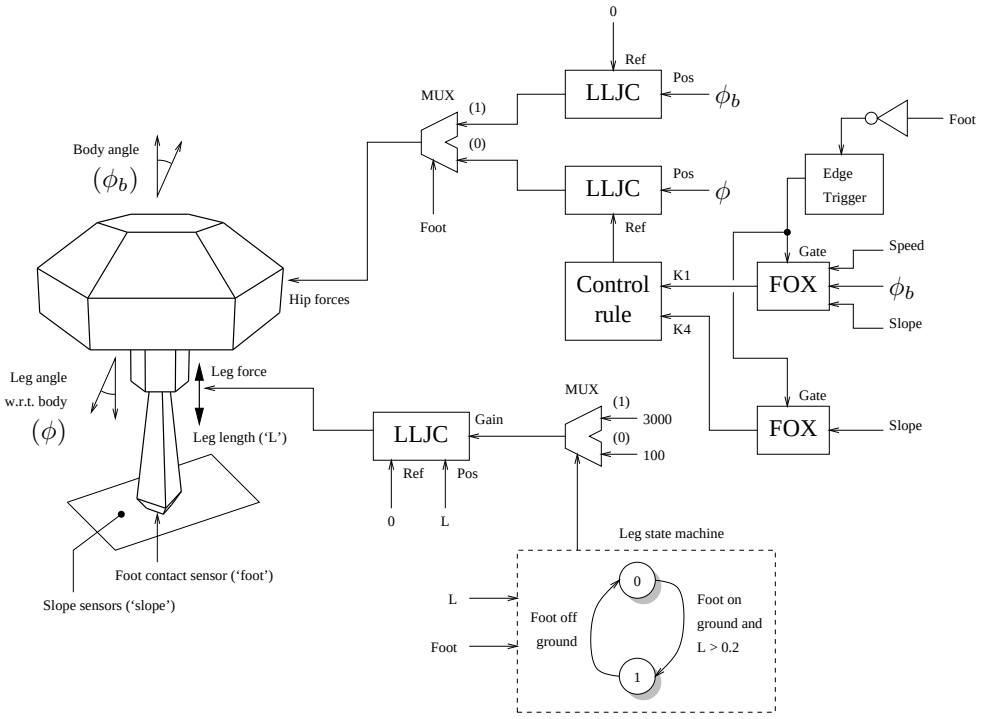


Figure 7.3: *A simplified view of the hopping robot's controller. Many details are omitted from this figure.*

where $k_1 \ldots k_4$ are constants. The k_1 term is the main foot positioning gain, to maintain the current speed. The k_2 term helps compensate for the body tilt when placing the foot. The k_3 term corrects speed errors, and the k_4 term adjusts the leg angle on a slope for better balance (this is necessary because the foot has a significant width).

7.5.3 The learning controller

Selection of the four control parameters to achieve stable control is tricky, but it is certainly possible, either from experiment or from an analytical understanding of the robot's motion. However, the task here is to use FOX to determine some of them from experience and (in principle at least) save the designer from finding them. This has other advantages: the controller is potentially better adapted to this nonlinear system than a pure linear controller (for example the optimum value of k_1 varies with speed), and the controller is more context sensitive, that is better adapted to different environments (different ground slopes in this case). Figure 7.3 shows how two FOX modules provide k_1 and k_4 (good values for k_2 and k_3 were found experimentally to be 0.8 and 0.1).

The k_1 FOX used overshoot training with output limiting. Its error signal was:

$$e \quad = \quad -(\text{desired\_speed} - \text{speed}) \cdot \text{sgn}(\text{speed}) \tag{7.3}$$

The sgn$(\cdot)$ functions returns ± 1 depending on the sign of its argument and is used to ensure that k_1 will be positive for motion in both the positive and negative directions. Overshoot training is essential in this case to prevent k_1 increasing without bound whenever the desired speed is changed. The output of the FOX is sampled and held stationary each time the foot leaves the ground. This makes the control problem slightly more difficult (the FOX cannot correct k_1 during the flight of the foot) but it prevents the leg from undergoing an oscillatory "hunting" behavior where it tries to seek the correct position (as the value of k_1 varies) during foot flight. The FOX is gated by an edge triggered version of the foot signal, that is the gate is only turned on at the instant when the foot leaves the ground (see section 5.9.2 for a definition of the gate). This is essential for training because the FOX output can only affect the system at the gated times. A critically damped eligibility profile is used with $t_{\text{max}} = 1$ (refer to Appendix E for the definition of t_{max}).

The k_4 FOX is configured similarly, except that its error signal is designed to only provide correction along slopes:

$$e \quad = \quad -(\text{desired\_speed} - \text{speed}) \cdot \text{sign}(\text{speed}) \cdot \text{slope} \tag{7.4}$$

7.5.4 Results

In the early stages of training the robot has not learned to correct its speed when it goes too fast, and so it trips and falls over frequently (figure 7.4). A movie of four out-takes from training was created (see movie `hopper1.mpg`) which shows the robot falling over in progressively later stages of its desired trajectory (right at the start, turning the first corner, going up the ramp, and going down the ramp). It shows that the robot must learn independently how to move in all four compass directions as well as up and down slopes.

After 20 training iterations (each iteration corresponds to falling over once) the robot has acceptable performance. Figure 7.5 shows four images from one attempt to follow the trajectory (also see movie `hopper2.mpg`).

The robot can hop along at a constant speed on the flat, it can turn corners, and climb and descend the ramp without falling. Its descent down the ramp is rather too fast, and in fact the controller described

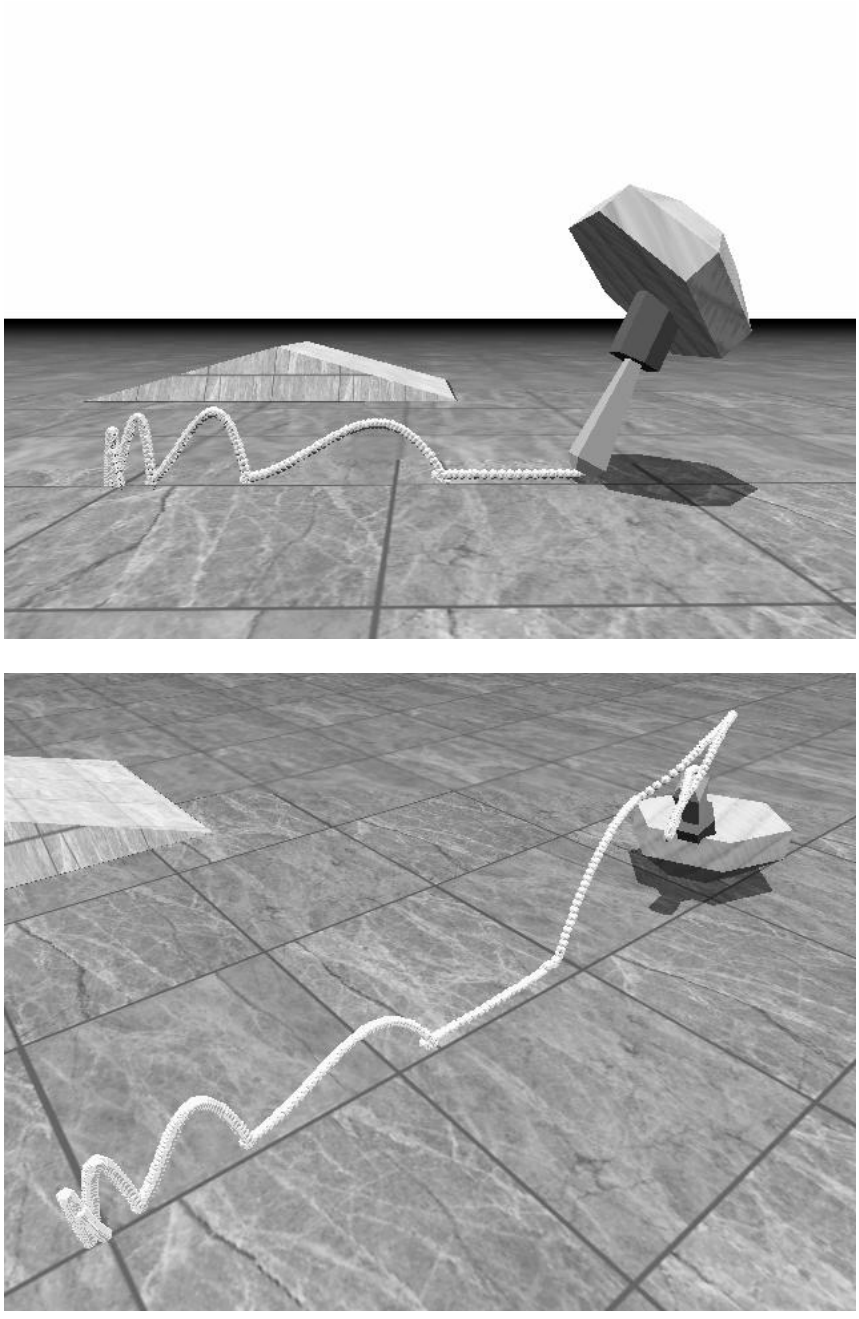


Figure 7.4: *The monoped hopping robot in the early stages of training. The robot has not learned to correct its speed when it goes too fast, so as a result it trips and falls over.*

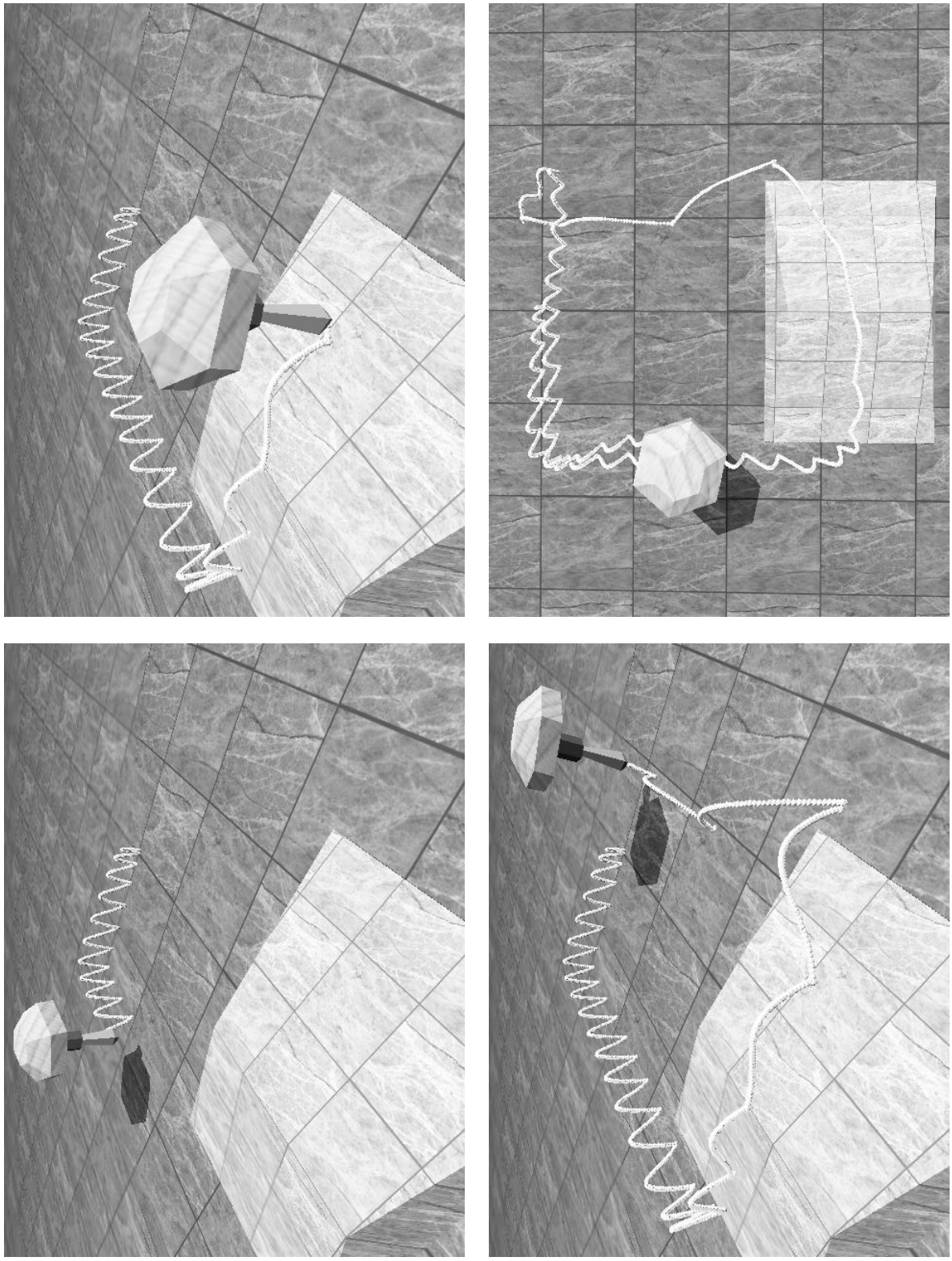


Figure 7.5: *The monoped hopping robot after 20 iterations of training (each iteration corresponds to falling over once). The robot follows a roughly square path that takes it up and down the ramp. The images sequence is bottom left, top left, bottom right, top right.*

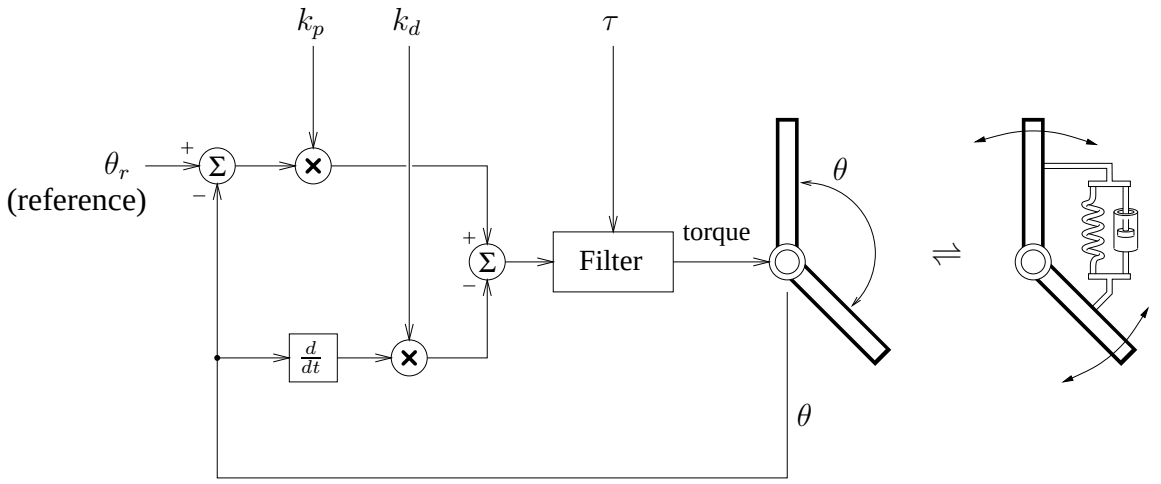


Figure 7.6: *The low level joint controller (LLJC), which is equivalent to having a spring-and-damper on the joint with an adjustable spring set-point.*

here never learns to limit the down-slope speed. This is an important point: the FOX modules do not have the freedom to optimize the entire motion of the robot. Instead they are just allowed to control two parameters of a controller with a fixed complicated structure. The FOX modules are only capable of changing the robot's behavior in limited ways. In general this can be beneficial, in the sense that training can never override behaviors that the designer has already determined to be useful. In other words, the trade off is between a complicated controller parameterized by FOXs with simple error functions, or a simple controller parameterized by FOXs with complicated error functions.

The hopping robot experiment provided a useful starting point for designing the more complicated biped controller. It introduced the idea of control with state machines and using different control modules in different parts of the robot's trajectory. It has also demonstrated that learning controller parameters is an effective alternative to explicitly learning an entire control force profile.

7.6 System components

Now a more detailed look will be taken at the components that can be used in robot controllers. The controllers for the hopping and walking robots are combined continuous and discrete dynamical systems. Internal state is maintained by discrete state machines, or timers, or less frequently by integrators. Different modules implement different behavioral tasks, and the modules pass information to each other to coordinate their activities.

At the lowest level, most actuators are controlled using 'LLJC' (low level joint controller) modules which implement simple proportional plus derivative joint-position control. The controller of figure 7.6 is used, which provides a simple model of the muscles and the most important spinal cord reflexes, in accordance with the conclusions of Latash [65]. It is equivalent to having a spring-and-damper on the joint with an adjustable spring set-point.

Instead of a single global performance error there are a number of local error values that each apply to a small piece of the problem. For example, errors may be formulated to keep the robot body level when one leg is on the ground, or to make sure the other leg is lifted far enough to clear any obstacles

when the controller is in a certain state.

Many types of controller parameters can be learned by FOX modules. For example:

- Coefficients for LLJC modules.
- Desired joint positions at various stages of locomotion, or the end positions for movements.
- The timing of state changes.
- The selection of state changes, that is, the potentiation of the next state that will be entered.

The last item (state potentiation) has been shown to be useful in the author's previous work [115] for high level behavioral selection. FOX modules can co-operate to control a single parameter (if there are multiple constraints on a single value), but in many instances the FOX modules act independently. For this to work well, the variables controlled by each FOX must be as independent as possible, so that minimization in one variable does not affect the others. In practice this is rarely possible, so there is often a lot of undesired interaction between the various learning processes. This can make the system's performance worse in the short term, but in a well designed system things will usually settle down and performance will improve.

The controllable parameters will differ in the level of influence they have on the system. Low level control parameters will contribute directly to the actuator outputs (the joint forces). Higher level control parameters will undergo additional processing by the controller (for example, the joint position references). It is generally preferable to control high level parameters, because the additional controller processing can generally tailor the system's nonlinear dynamics so that the eligibility profile concept is more applicable (this may be more difficult if low level parameters like force are being controlled). In general, nonlinear state machine based controllers stretch the FOX theoretical model to its limit—in many cases it will just be assumed that FOX control will work, without any rigorous justification. However, it is anticipated that additional "internal" feedback controllers will usually be incorporated into the system to provide appropriately modelable dynamics.

Another reason for FOX modules to control high level parameters is that this can result in far more robust control. This is because when the system enters an unforeseen state (one previously untrained) and the FOX outputs are zero, the controller still has a chance of providing an adequate response if the parameter value of zero results in some default behavior. If the FOX controls a parameter that is too low level (such as position or force) then a zero output will be inappropriate in almost every situation, and system failure during learning will be much more frequent.

Many parameters will only influence the system for part of the time (the rest of the time the system will be in a state where the parameter is not even considered). In these cases the FOX eligibility driving force must be gated, as explained in section 5.9.2. If this does not happen then the FOX may learn inappropriate things. However, an alternative in some situations is to gate the FOX error signal instead.

The FOX error signals, error function and learning rates have to be chosen with care, and some experimentation is usually needed before a working combination can be found. The FOX eligibility profile may be difficult to create for a highly nonlinear robot-plus-controller system. But in most cases it has been found that a second order critically damped profile with an estimated $t_{\max}$ value in the range $0.1\text{s} \ldots 2\text{s}$ is sufficient (see Appendix E for the definition of $t_{\max}$).

A design compromise must be made when choosing the inputs to each FOX. Fewer inputs mean that the FOX will be able to generalize its training to a wider range of unforeseen situations. But more inputs mean that the FOX will be able to individually tailor its output to a wider range of specific situations.

If the controller had no internal state, that is if it was purely sensor driven, then a given sensor picture would always result in the same actions. It is usually more useful to have one or more state

machines or timers inside the controller so that it can have an internal notion of its current "plan" which is independent of the outside world. Transitions between states can be triggered when some function of the sensors climbs above a threshold, or when a time limit for the current state has been reached.

When designing a controller a choice must be made between state-dependent (feedback) control and state independent (feed-forward) control. A feedback controller depends completely on its sensors to determine state changing decisions and actuator outputs. A feed-forward controller makes state changing and actuator output decisions according to some pre-arranged schedule. In practice both methods are used together, although a particular controller may be biased towards one or the other. Feed-forward control allows a pre-arranged behavior to be played out—some parameters of this behavior are learned so that it will be correctly mapped to the environment. But feed-forward control relies on the system being in a particular known state (or limit cycle) and it will not work well otherwise. With feedback control it can be trickier to get a particular sequence of movements, but the controller is usually a lot more robust to body states outside those anticipated by the designer. The hopping robot was largely feedback-based. The biped robot which will be described in the next section has feed-forward components.

7.7 Making a biped robot walk

7.7.1 A short review of biped locomotion

The problem of biped robot locomotion has received a lot of attention. This is partly due to the great difficulty of the problem (and therefore its high research value), partly because of the desire to understand the principles behind human locomotion, and partly due to the presumed superiority of bipeds over quadrupeds and hexapods, and wheeled or tracked vehicles (after all, human walking is highly adaptive and versatile).

Despite this, the problem is largely unsolved. Although there are now many real and simulated bipeds that can walk and run, none of these systems can survive for long in unstructured environments without falling down. Although more than 60 different climbing and walking machines developed in research laboratories and universities have been cataloged, industrial applications are emerging very slowly [106]. No *bipeds* have yet been deployed in any practical applications, although various robot with four or more legs have been.

Legged robots are either statically or dynamically stable [18]. Statically stable robots will retain their balance if they stop moving at any time, thus they present a relatively easy control problem. They typically have four or six legs but may be bipeds with large feet. Dynamically stable robots are only stable in a limit cycle that repeats once each stride. They are much more difficult to control, but (in theory at least) can provide more versatile locomotion.

From the point of view of current control theory the dynamically stable locomotion problem is not overwhelming. Indeed, several analytical studies have been made which show how biped walking can be achieved. For example, Kajita and Tani have studied dynamic biped walking using a "linear inverted pendulum" model for leg movement. Their scheme uses linear control approximations, and has been tested on a real six degree-of-freedom robot [56]. Similarly, Pannu *et al* have tested the analytical μ-synthesis control approach on one leg of a walking robot [93].

Raibert and his colleagues have built dynamically stable running monopeds, bipeds, and quadrupeds [18]. These robots all achieved locomotion by bouncing on springy legs. Their control systems achieved stability by decoupling the problem into three parts: (1) controlling hopping height through the leg actuators, (2) controlling forward speed by correctly positioning the foot at touchdown, and (3) controlling the body attitude by applying torque at the hip while the foot was in contact with the ground [103]. Finite state

machines were used to switch between the different control laws required in each mode. Three dimensional hopping and running was achieved by decomposing the robot motion into planar and extra-planar parts [104]. Recently this work has been extended to make the locomotion more robust in real-world environments: Boone has implemented reflex-like actions to help biped robots recover from slipping and tripping [17].

Almost all current implementations rely on a well structured environment (usually just a hard, flat walking surface). Most current implementations can not cope with uneven terrain, slopes, varying contact friction, and obstacles—that is, they are not very adaptive. This is because most walking machines to date do not learn: their control parameters are computed analytically or adjusted by hand. But there are a few exceptions. Miller has achieved balanced walking in a real ten axis biped robot using CMACs to learn some controller parameters [83, 84]. The controller contains a hierarchy of gait oscillators, PID controllers, and CMAC networks. His scheme does not require a detailed robot dynamical model, but the robot can only take short steps without falling over. Lin and Song have also used CMAC neural networks for the purely kinematic control of legged robots [70]. Stitt and Zheng have developed a method that generates biped gaits suited to varying ground slopes, based on "distal supervised learning". Their technique requires a forward model of the robot's dynamics, and converts information about stable gait on a flat floor to rules for climbing and descending slopes [116].

Many controllers have been designed using semi-biological principles. For example, Crawford has designed a hierarchical control structure that uses radial basis function networks, for the control of a human platform diver [30, 29]. It is proposed that systems with many degrees of freedom can be controlled with a hierarchical network of simple single-joint learning controllers. Hallam *et al* go even further with their neuroethological approach in which a neural network with quasi-realistic synapse modification is used to control a robot [41].

The difficulty of the biped locomotion problem makes it a good test bed for new control theories. Failure in biped locomotion is generally catastrophic, as no current system has the coordination to pick itself up once it has fallen down. Thus one can always gauge the success of biped control schemes in an ad-hoc manner by asking "how long can it walk without falling over?".

The biped controller presented here is more adaptive than most, but it is still unsuited to any practical application. The chief value of these experiments is to test the value of FOX-based controllers in complex robot problems.

7.7.2 The problem

Experiments were performed on the simulated biped robot shown in figure 7.7. The biped has one significant mechanical deficiency compared to a human—it has no lower back / hip flexibility. The biped's mass parameters are given in table 7.2.

The robot was given three tasks:

1. Learn to walk in a straight line with a steady gait.
2. Learn to walk in a circle (to test the robot's ability to change direction).
3. Learn to walk up and down a ramp (to test the robot's ability to adjust its gait to the slope of the ground).

7.7.3 Feed-forward controller structure

A feed-forward walking approach similar to Laszlo *et al* [64] was used. In that study a walking biped (with 19 degrees of freedom) was simulated using limit cycle control, with the intention of rendering

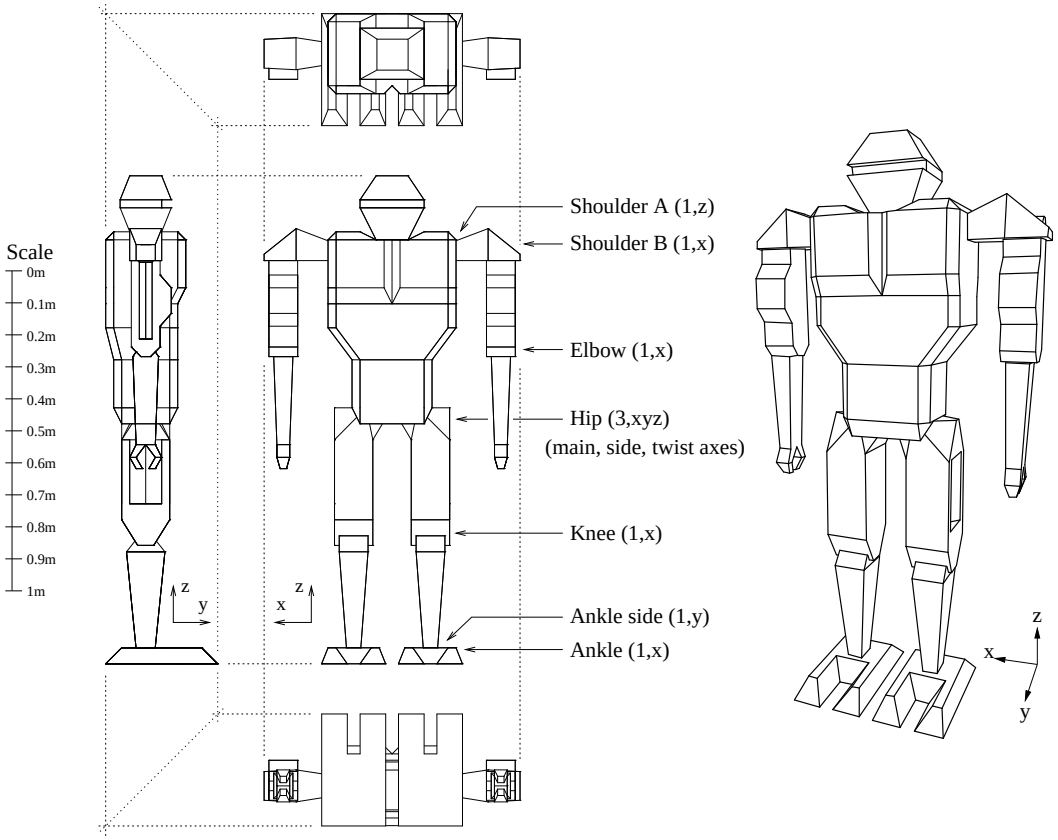


Figure 7.7: *The structure of the simulated biped robot. The robot has 18 degrees of freedom spread over seven joints on each side of the body. Each joint is labeled with the number of degrees of freedom it has and the axes (x,y,z) it can rotate through. The hip joint is a ball-and-socket which is controlled along the main (x), side (y) and twist (z) axes. The other joints are simple revolute (hinge) joints.*

Link	Mass (kg)	Rotational inertia (kg m^2)		
		x-axis	y-axis	z-axis
body	30.0	0.9125	1.15625	0.1696
upper leg	4.0	0.2208	0.2181	0.0123
lower leg	2.5	0.0047	0.1038	0.1051
foot	0.5	0.08	0.018	0.065
shoulder link	1.0	0.014	0.0017	0.014
upper arm	2.0	0.0014	0.083	0.083
lower arm	1.5	0.0009	0.0617	0.0617

Table 7.2: *The mass parameters of the simulated biped robot.*

realistic looking walking for computer graphics applications. Stereotyped open-loop (feed forward) periodic walking motions were generated using a finite state machine (they were not strictly open-loop as PD controllers were used at each joint, but there was no global feedback to hold the system on its desired trajectory). By themselves these motions did not result in stable walking, as sensor feedback was not used. The walking motion was stabilized by adding closed loop feedback. An off-line hill-climbing learning method was used to find the control parameters that kept the walking motions on a limit cycle (so that falling-over perturbations were automatically corrected for). The biped's direction, speed and stride rate were controllable.

Figure 7.8 shows the internal structure of the controller used in this experiment. Each of 18 joints (six in each leg and three in each arm) are controlled by an LLJC module. A variety of high-level modules control various aspects of the biped's motion. Figure 7.9 shows a stick-figure representation of how each controller module contributes to the biped's motion. Each module produces desired reference values for one or more joints. These references are combined together in the RC (reference computation) units which compute the final joint references (see table 7.4 for details, and table 7.3 which defines the symbols used). Eighteen unknown parameters are learned by FOX modules (eight which are duplicated in each leg and two which are global—see table 7.5). Second order critically damped eligibility profiles are used (defined by the parameter $t_{\max}$, see Appendix E). The values of $t_{\max}$ were selected somewhat arbitrarily, based on a mixture of intuition, guesswork and experimentation.

Note that there a *many* possible controller structures, and also a great number of potentially learnable parameters. The selection used in this experiment is not claimed to be the best: the purpose of the experiment is merely to demonstrate FOXs usefulness in this kind of system.

The operation of the biped after training is shown in figure 7.10, figure 7.11, and in movie `walkt2.mpg`. The robot can successfully walk with a steady gait, change its stride length, change direction and climb and descend slopes.

The controller operation will now be described (note that many details have been omitted for the sake of clarity). It will be shown how each controller module separately influences the behavior of the system, by showing what happens when that controller is removed.

A variety of sensors are available to the controller: 18 joint angular positions and velocities, the absolute body orientation along three axes (ϕ, θ, ψ), contact forces at the heel and toe, and the slope a short distance in front of the robot. A simple state machine for each foot (not shown) determines when a solid contact has been made with the ground.

The principle goal of the controller is to keep the body upright while maintaining the desired forward speed and zero sideways deviation from the desired path. The basic motions of walking are generated by

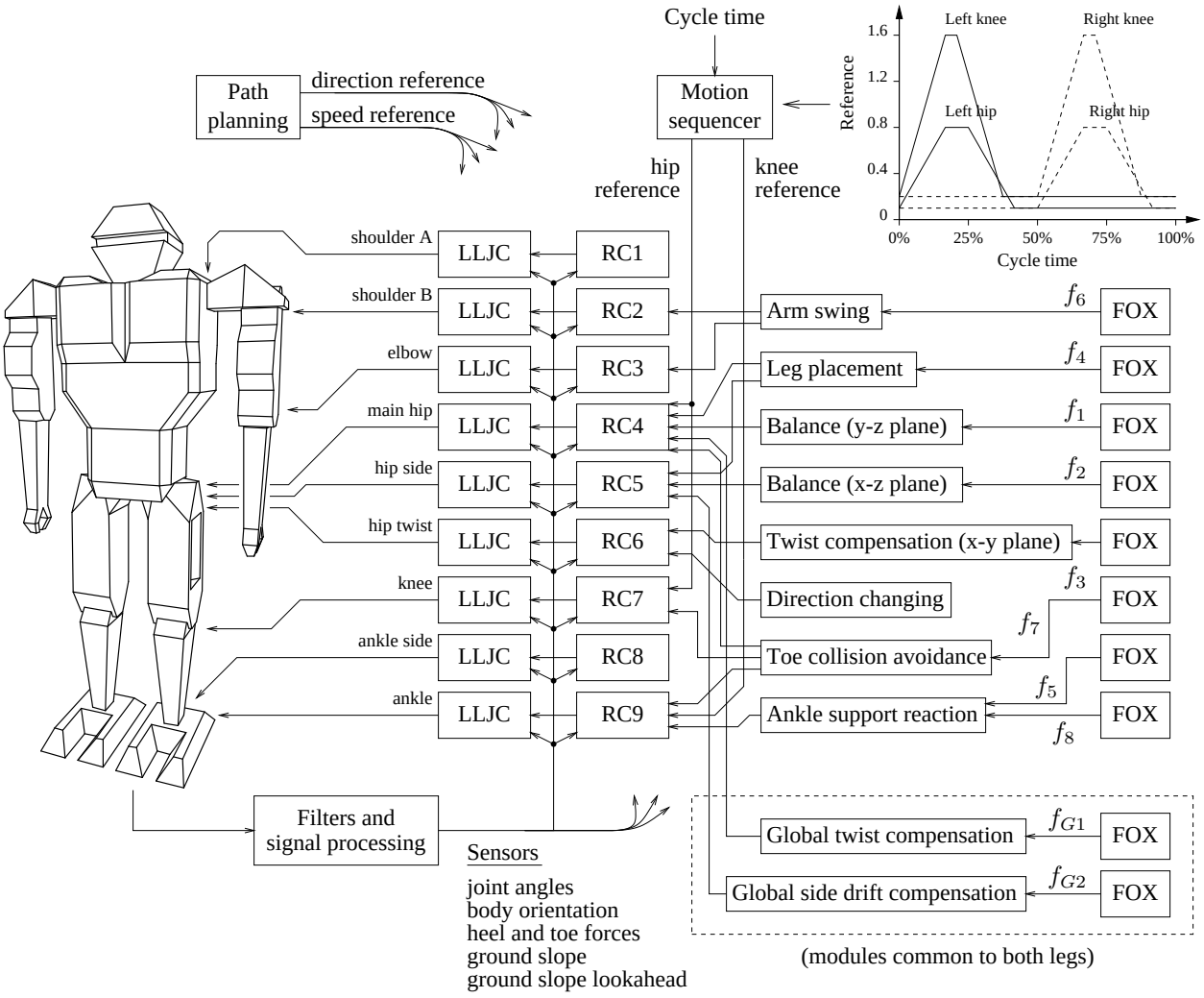


Figure 7.8: *The structure of the "feed-forward" controller for biped walking. Note that many details are omitted from this diagram.*

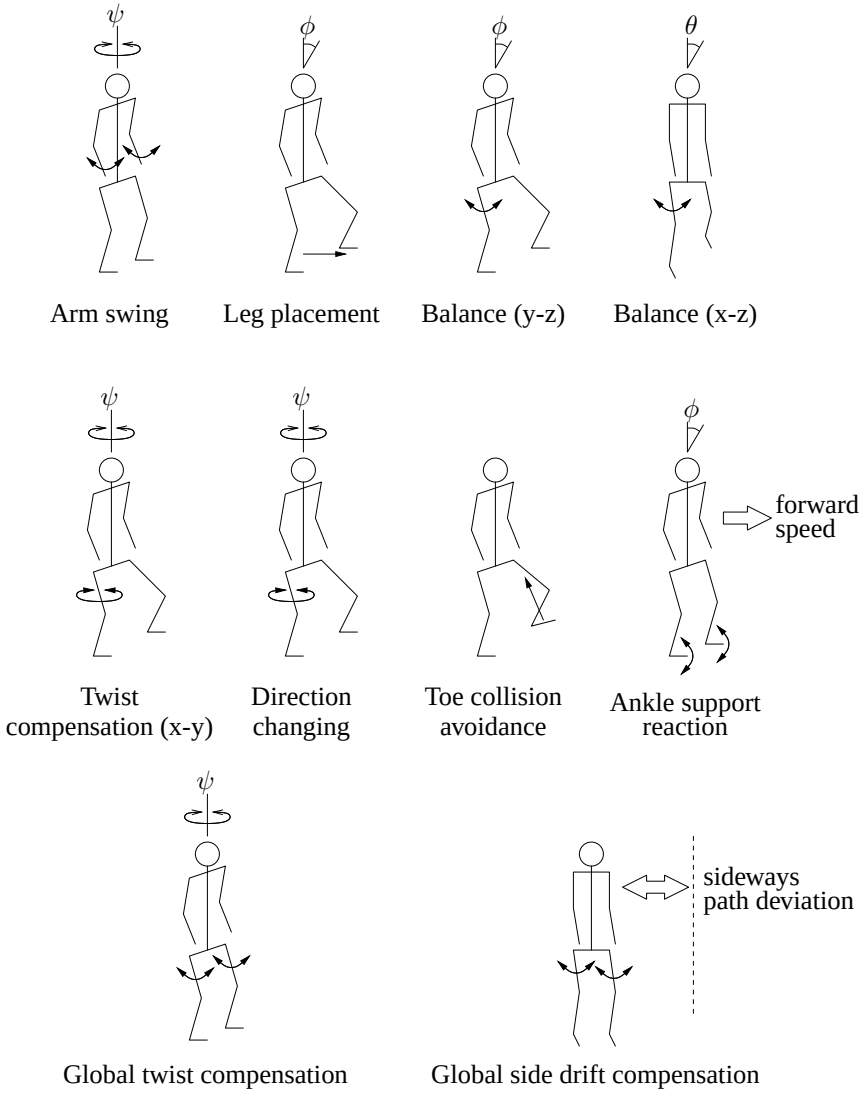


Figure 7.9: *A stick-figure representation of how each controller module contributes to the biped's motion. In each figure the arrows show the controlled joint, and the corresponding error variable is indicated (either ϕ, ψ, θ, forward speed or sideways deviation).*

Symbol	Definition
	Motion sequencer
R_1	Hip reference
R_2	Knee reference
	Desired values
r_1	Desired value of ϕ (forward tilt)
r_2	Desired forward speed
r_3	Desired value of ψ (direction)
	Errors
e_1	ϕ error $= r_1 - \phi$
e_2	speed error $= r_2 - s_3$
e_3	sideways path deviation $= -s_6$
	Sensors
s_1	Is foot touching the ground? (boolean)
s_2	Is the opposite foot touching the ground? (boolean)
s_3	Forward speed (m/s)
s_4	Is the foot firmly on the ground? (boolean)
s_5	Is the opposite foot firmly on the ground? (boolean)
s_6	Sideways path deviation (m/s)
s_7	Contact force at heel (N)
s_8	Contact force at toe (N)
s_9	Slope of ground just in front of robot (radians)
ϕ	Body angle in the y-z plane (forward tilt)
θ	Body angle in the x-z plane (sideways tilt)
ψ	Body angle in the x-y plane (twist)
	FOX outputs
f_1	Reference adjustment for main hip
f_2	Reference adjustment for hip side
f_3	Reference adjustment for hip twist
f_4	Reference adjustment for main hip (2)
f_5	Reference adjustment for ankle
f_6	Reference adjustment for shoulder B
f_7	Reference adjustment for hip,knee and ankle
f_8	Reference adjustment for ankle
f_{G1}	Global reference adjustment for main hip (3)
f_{G2}	Global reference adjustment for hip side (2)
	Miscellaneous
sign	$+1$ on the left, -1 on the right
t_c	cycle time $(0 \ldots 1)$
a_s	action sine $(-1 \ldots 1)$
a_c	action cosine $(-1 \ldots 1)$

Table 7.3: *Definitions of the symbols used in the walking biped controller (see table 7.4). The values $f_1 \ldots f_8$ and 'sign' are duplicated for each leg.*

Joint	Reference (radians)
Shoulder A	0
Shoulder B	$-R_1(0.7 + f_6) + 0.15$
Elbow	$0.5 + 0.5R_1$
Hip (main)	$R_1 - r_1 + (\text{not } s_1)(1.5e_1 + 0.8s_3) + f_1 s_4 + \text{sign } f_{G1}(s_1 \text{ and } s_2) + (\text{not } s_1)(f_4 + f_7)$
Hip (side)	$f_2(s_4 \text{ and not } s_5) + f_{G2}(s_1 \text{ and } s_2) + (\text{not } s_1)(-0.1 \text{ sign} - 2\theta - 0.3s_6)$
Hip (twist)	untwist_ref $+ f_3 s_4$
Knee	$R_2 - 2(\text{not } s_1) f_7$
Ankle (side)	$2\theta + 0.5\dot{\theta} + 0.3e_3$
Ankle	$(s_1 \text{ and } R_1 \leq 0.15)((-0.12 - 0.003(s7 - s8)) + 0.3e_1) - (\text{not } s_1 \text{ or } R_1 > 0.15)R_1 - (\text{not } s_1) f_7 + s_4(f_5 + f_8)$

Table 7.4: *The joint references computed by the RC modules for the walking biped (incorporating all controller modules and FOX-learned parameters). Note that some details are omitted. For definitions of the symbols used, see table 7.3.*

FOX	Inputs	Gate	Error	$t_{\max}$ (s)	Learning rates
f_1	a_s, a_c, s_3	s_4	$-e_1$	0.4	$\alpha_2 = 0.0005, \beta = 0$
f_2	a_s, a_c, s_3	s_4 and not s_5	θ	0.2	$\alpha_2 = 0.005, \beta = 0.00001$
f_3	a_s, a_c, s_3	s_4	$\dot{\phi} s_4$	0.4	$\alpha_2 = 0.005, \beta = 0.00001$
f_4	a_s, a_c, s_3, s_9	not s_1	e_1	0.5	$\alpha_2 = 0.0002, \beta = 0$
f_5	a_s, a_c, s_3	s_1 and $R_1 \leq 0.15$	e_2	0.05	$\alpha_2 = 0.00005, \beta = 5 \times 10^{-7}$
f_6	a_s, a_c, s_3	1	$\dot{R}_1 \dot{\psi}$	0.2	$\alpha_2 = 0.0005, \beta = 0$
f_7	a_s, a_c, s_3	not s_1	s_1 and $t_c < 0.3$	0.3	$\alpha_2 = 0.05, \beta = 0.005$
f_8	a_s, a_c, s_3, s_9	s_1 and $R_1 \leq 0.15$	e_1	0.5	$\alpha_2 = 0.0005, \beta = 5 \times 10^{-6}$
f_{G1}	a_s, a_c, s_3	s_1 and s_2	$\dot{\psi}(s_1 \text{ and } s_2)$	0.2	$\alpha_2 = 0.001, \beta = 0.00001$
f_{G2}	a_s, a_c, s_3	s_4 and s_5	$-e_3$	1.0	$\alpha_2 = 0.01, \beta = 0.001$

Table 7.5: *The inputs and parameters of the FOX modules used in the walking biped controller. In all cases, overshoot learning with output limiting is used, with $\alpha_1 = 0.01\alpha_2$. Second order critically damped eligibility profiles are used, defined by the parameter t_{max} (see Appendix E).*

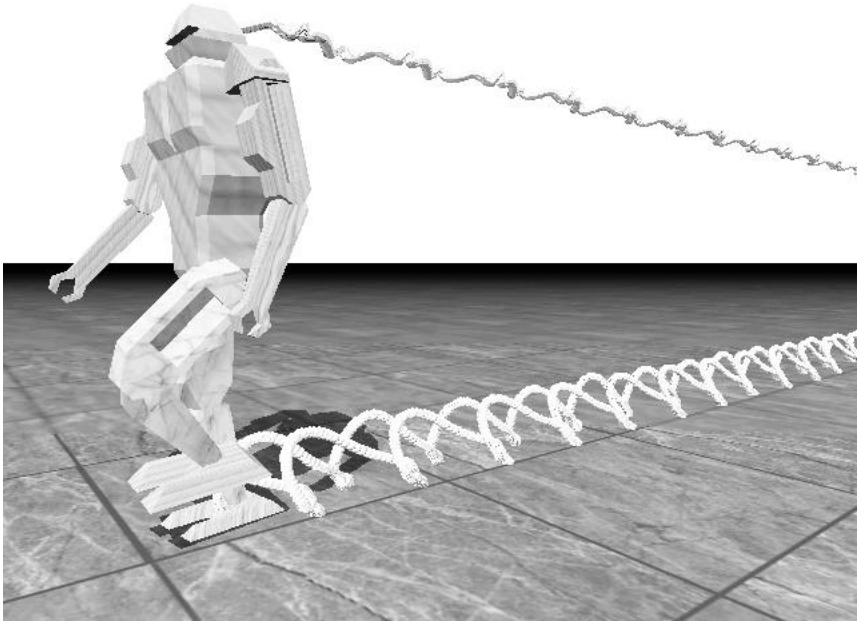


Figure 7.10: *Successful biped walking. Note that trails are rendered from the top of the head and from both ankle joints.*

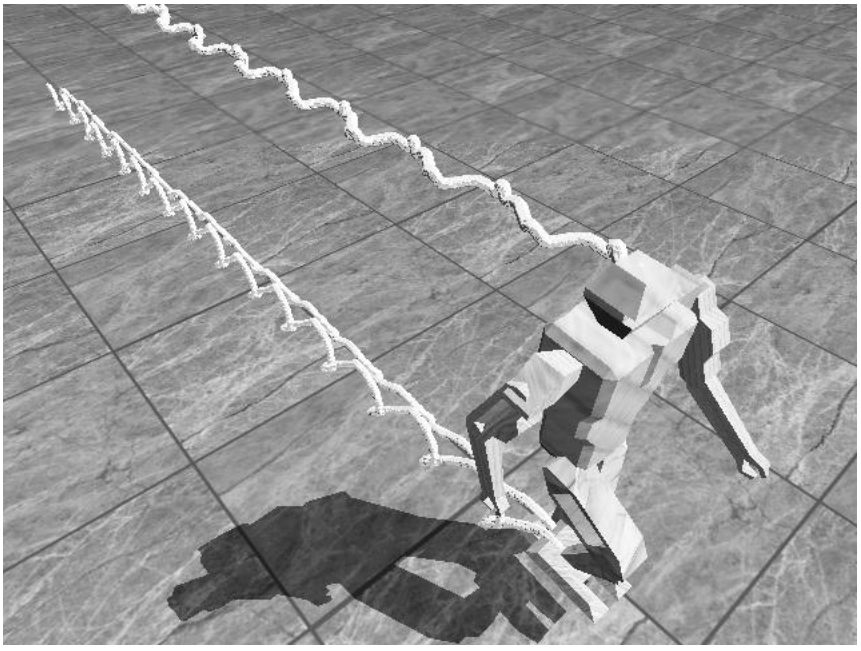


Figure 7.11: *Successful biped walking—another view.*

the motion sequencer which outputs periodic preprogrammed hip,knee and ankle joint references (at a fixed frequency of 0.75Hz) which extend and contract the leg. Without the other controller modules the motion sequencer can drive the robot through only a couple of steps before it falls over (see figure 7.13, movie `walk1.mpg`).

All FOX modules used input resolutions between 80 and 200, and an n_a of 20. Each of the 18 FOX modules had 100,000 weights (far more than were really needed), for a total of 1.8 million weights in the entire system.

To achieve stable limit-cycle walking it is expected that the FOX outputs would have to vary in accordance with the cycle time. Thus the FOX inputs all include the variables a_s and a_c. These variables are sine and cosine functions of the cycle time: the point (a_c, a_s) goes around the unit circle once per cycle. They are a used instead of the cycle time itself to prevent CMAC local generalization discontinuities once per cycle. One advantage of the feed-forward mode is that the FOX modules need few other inputs to achieve sufficient specialization.

During the walking cycle each leg sees three different situations: (1) the foot is in the air, (2) the foot is on the ground while the other foot is in the air, and (3) both feet are on the ground. Each controller module is normally switched in during only one or two of these situations.

y-z and x-z balancing controllers: These balance the robot on one leg in the y-z and x-z planes (forwards and sideways tilt) when that foot is the only one on the ground. The main hip and hip side references are controlled, based on the current body tilts and x/y speeds. The f_1 and f_2 FOX modules adjusts these reference to fine tune the balance. Correct single-leg balancing tends to maintain or increase the current forwards/sideways speed, because the leg on the ground must be thrust outwards in the direction of travel to prevent the body angle from changing. This is demonstrated in figure 7.14 and movie `walk2.mpg`, which shows what happens when there is no hip side compensation.

Direction changing module: This tries to bring the biped's direction closer to the desired direction with each step. When the foot is placed on the ground and the other foot is lifted, the untwist_ref (table 7.4) angle is changed to bring the body around to the correct direction. When the foot is lifted again the reference is reset to zero. The f_3 FOX module adjusts the hip twist reference to stop the body from twisting when the foot is on the ground (see movie `walk3.mpg`).

Leg placement controller: This adjusts the main hip and hip side references when the foot is in the air, so that the leg position on foot impact will cause the long term balance to be maintained. The f_4 FOX module adjusts the main hip reference to ensure that this happens. Correct leg placement also tends to maintain the current speed. Figure 7.15 and movie `walk6.mpg` demonstrate the effect of leg placement.

Toe collision avoidance module: This prevents the foot from prematurely touching the ground during a step. The f_7 FOX module learns to cause a contraction of the leg (a hip-knee-ankle reference change) if the foot touches the ground during a part of the cycle when it is anticipated that the foot should be in the air (see movie `walk5.mpg`).

Ankle support reaction: This adjusts the ankle reference when there is a differential pressure between the heel and toe. This is done to help the biped keep its balance. For example, if the toe-pressure is greater than the heel-pressure the foot will be extended, increasing the toe-pressure in the short term but also hopefully pushing the body back to a more balanced position. This reaction also helps the biped to

push off with its rear foot in the moments before it takes a step with that foot. The set-point of this reflex can be adjusted to accelerate or decelerate the biped, by tending to tip it forwards or backwards. The f_5 FOX module adjusts this set-point to try and control the biped speed. The FOX's output is limited to prevent it from moving the set-point too far, which could cause over-balancing.

Another FOX module, f_8, adjusts the ankle reference to keep the biped upright. It is controlling the same parameter as f_5, so it needs to have a different eligibility profile to prevent the errors e_1 and e_2 from simply having an additive effect on the ankle's reference.

Arm swing module: This swings the arms to try and prevent the body from twisting during walking. The f_6 FOX module adjusts the gain of this reflex to get the best effect (see movie `walk3.mpg`).

Global twist compensator: This is common to both legs, and acts when both feet are on the ground to try and prevent body twist. It is essential, because the modules that control the main hip reference tend to "fight" each other when both feet are on the ground, causing a net torque to be applied between the feet which twists the body around. This twist can pull the feet out of their stable stance, causing the biped to slip. The f_{G1} FOX module changes the main hip references of both legs (using a different sign for each leg) to try and counteract this twisting effect.

Global side drift compensator: This is also common to both legs, and acts when both feet are on the ground to tilt the body such that the sideways desired-path deviation is minimized. The appropriate hip side reference adjustment is learned by the f_{G2} FOX module (see figure 7.16 and movie `walk4.mpg`).

The learning rates for all FOXs were chosen experimentally by observing the learned parameter. For example, if it did not grow fast enough, the learning rate was increased, and if it grew too large or started to over-train then the output limiting factor was increased.

7.7.4 Walking performance

As has already been seen, the biped can learn to walk with a steady gait. Figure 7.12 shows the training performance that was achieved with the system described here. A speed reference of 0.3 m/s and a constant direction reference were used. The time spent in each training iteration is shown (where each iteration ends when the biped falls over). After 220 iterations the biped is walking perfectly, with a total simulated training time of 74 minutes. The training time can be adjusted up and down by changing the learning rates. A compromise must be made: higher learning rates give faster learning, but the system is also more susceptible to learning abnormalities such as over-training and learning interference.

Notice that the biped's performance (iteration time) does not increase monotonically, as might be expected if this was a simpler optimal control system. Instead the biped goes through various training stages. In each stage a different failure mode is experienced (for example, falling to the side due to inadequate hip side compensation). Only a subset of learnable parameters experience a high training effect in each stage. Figure 7.17 shows a typical falling-over event near the start of training. A variety of such events are shown in movie `walk8.mpg`. Eventually the FOX modules have sufficient experience to keep the biped standing for extended periods of time, at which point the control parameters can be fine tuned to ensure a steady gait. Occasionally while the biped is walking it will get into an unanticipated configuration and fall over (figure 7.18), but these events are essential for the training experience to be comprehensive.

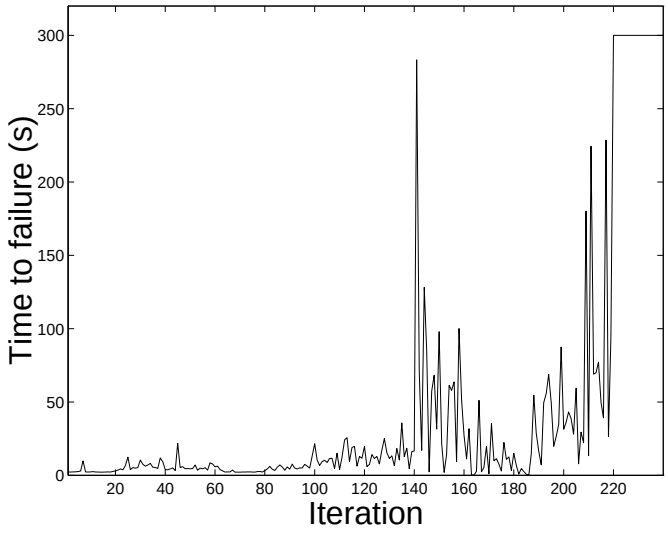


Figure 7.12: *The training performance for the walking biped. This shows the time spent in each training iteration, where each iteration ends when the biped falls over. After 220 iterations the biped walks perfectly (note the biped is reset after 300 seconds of successful walking). The total training time is 74 minutes.*

Once trained the robot was able to adapt its gait to speed references up to the reference used during training, as shown in movie `walkt2.mpg`. The biped was also able to follow an arbitrary path, simply by changing the direction reference and path deviation calculation (see figure 7.19 and movie `walkt2.mpg`).

The biped was also tested on its ability to walk up and down slopes. The test environment is shown in figure 7.20. After training the biped successfully crossed the ramp, as seen in figure 7.21, figure 7.22 and movie `walkt2.mpg`. The biped had to learn to adjust its gait to suit each slope separately. This is shown in figure 7.23, where the biped has successfully climbed the up-slope of the ramp but trips over on the flat top because it has not yet had flat-ground experience.

The effect that training has on the biped motion is subtle but significant. Figure 7.24 shows some biped joint angles over three stepping cycles, before training. The same thing after training is shown in figure 7.25. The general appearance of both plots is the same, that is the biped makes almost the same motions in each case. But on closer inspection there are many fine differences, which are required for stable walking.

Figure 7.26 shows three trained FOX outputs over three stepping cycles. The complex form of each FOX output has been acquired through training via the need to compensate as much as possible for the perturbing effects of the rest of the system (which includes the other FOX modules). In effect, the form of each signal is specially constructed to *anticipate* what the biped might be doing wrong and compensate for it before it has a chance to manifest itself. Also notice that the activity of each FOX output is concentrated in the region where its gating signal is nonzero.

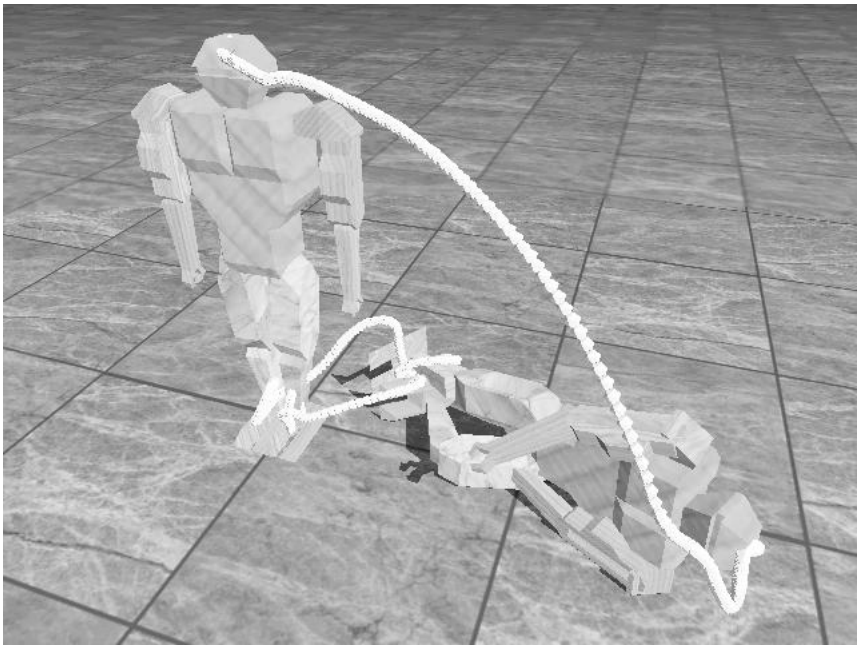


Figure 7.13: *Biped walking, where only the motion sequencer is used in the controller (no other controller modules are active). The biped makes stereotyped stepping movements and falls down immediately.*

Figure 7.14: *Biped walking, without hip side compensation. The biped sways from side to side as it walks (the trail left by the head has a large side-to-side variation).*

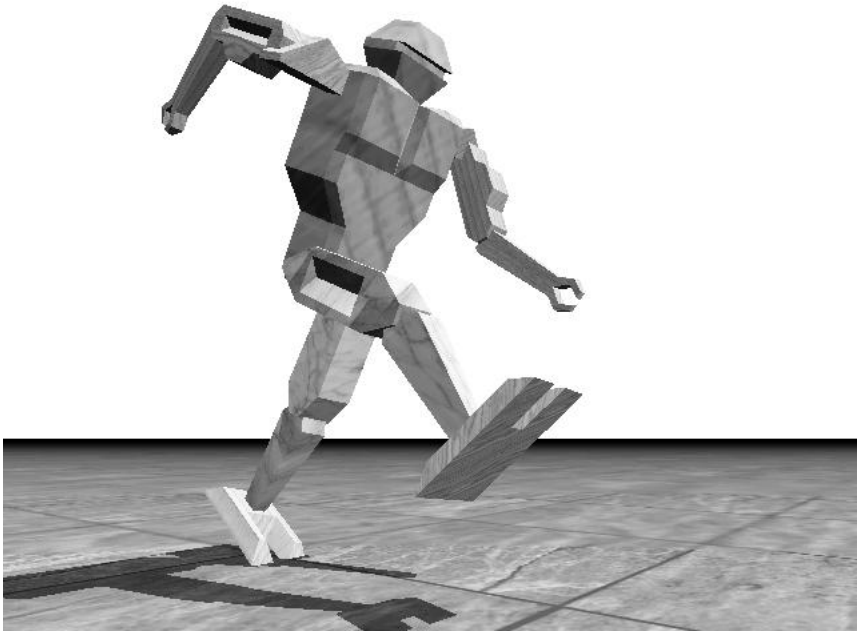


Figure 7.15: *Biped walking: this illustrates leg placement (the effect of f_4). When the foot is off the ground, the leg is placed to try and prevent a large body angle or forward speed.*

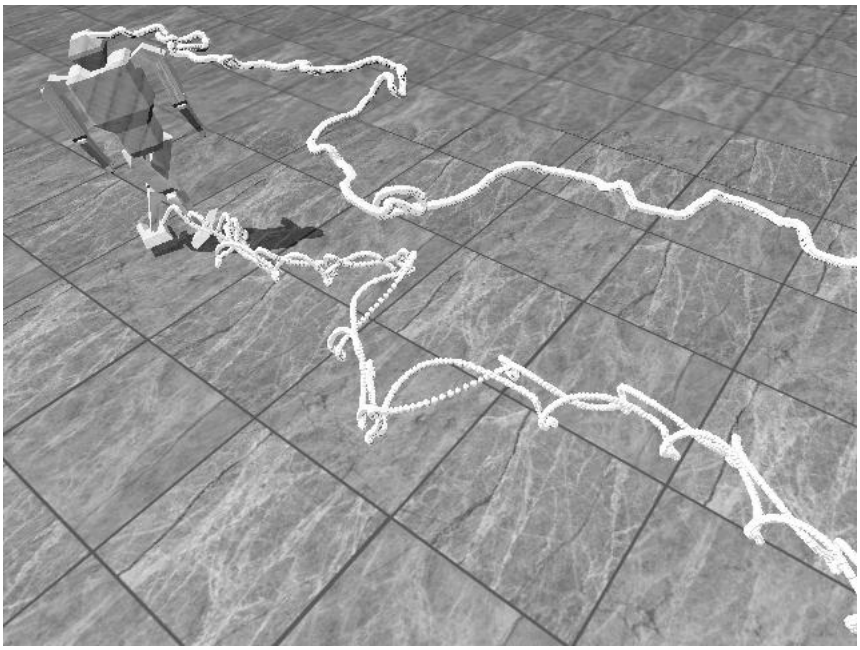


Figure 7.16: *Biped walking, without global side drift compensation. The biped strays from the desired path and becomes unstable trying to get back on it.*

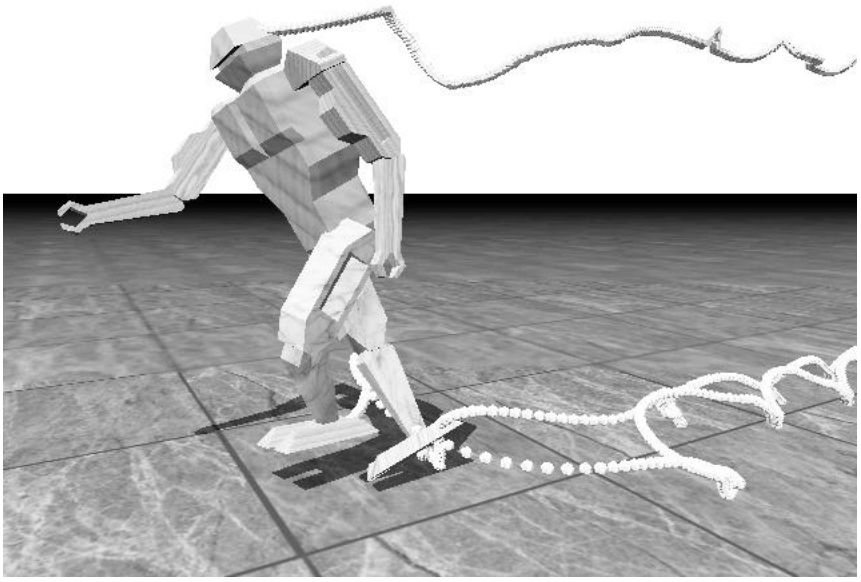


Figure 7.17: *Biped walking: this is a typical falling-over event near the start of training.*

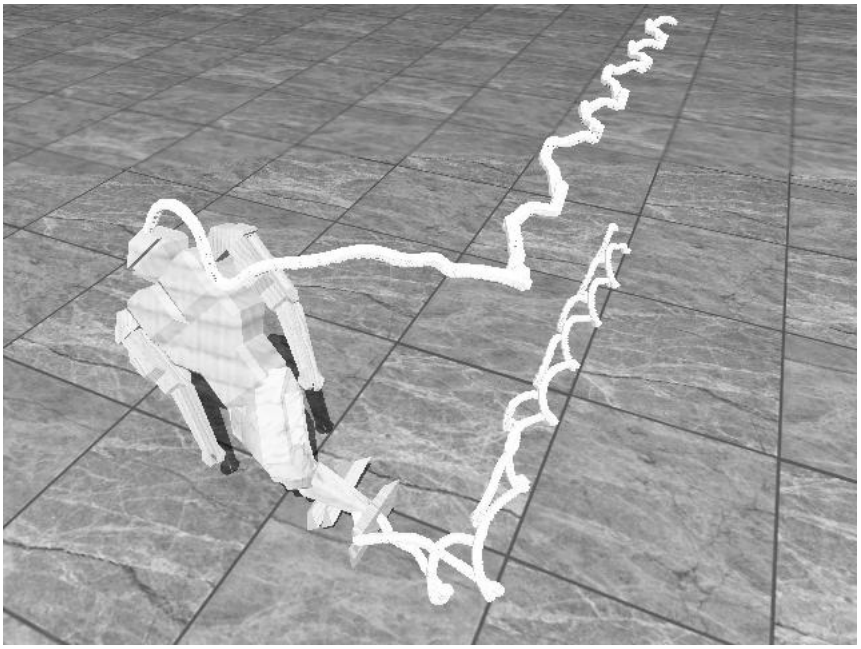


Figure 7.18: *Biped walking: this is a typical falling over event during training. The biped walks well for a while, then suddenly falls over.*

Figure 7.19: *Biped direction changing: the biped walks in a circle around the marker.*

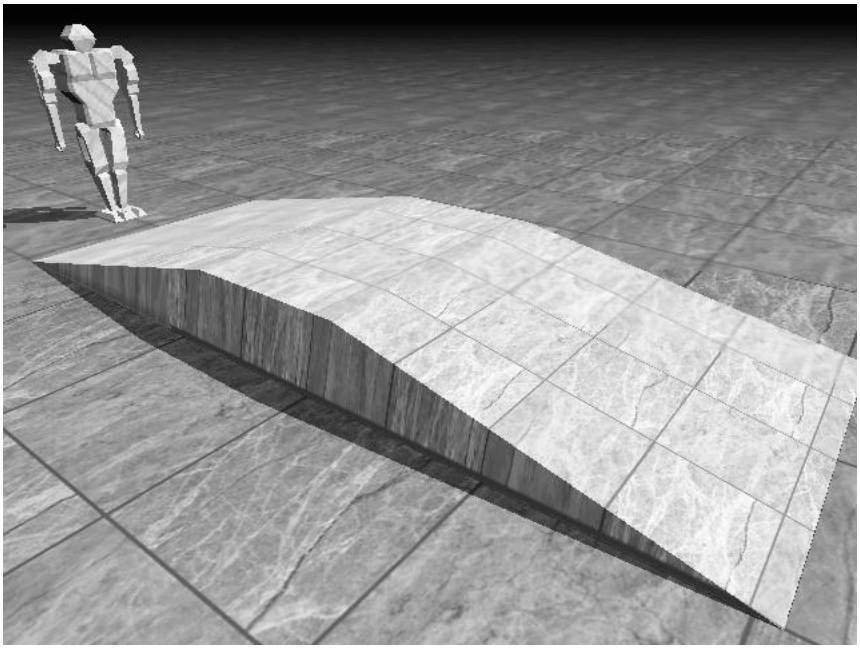


Figure 7.20: *Biped walking on slopes: this is the test environment, with a ramp the biped must climb and descend. The biped is in its starting state.*

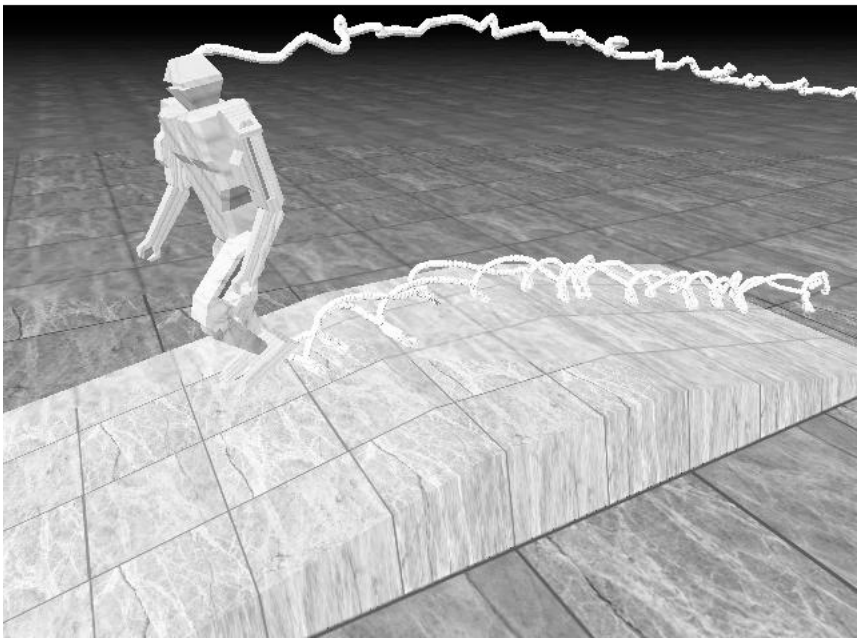


Figure 7.21: *Successful biped walking over the ramp.*

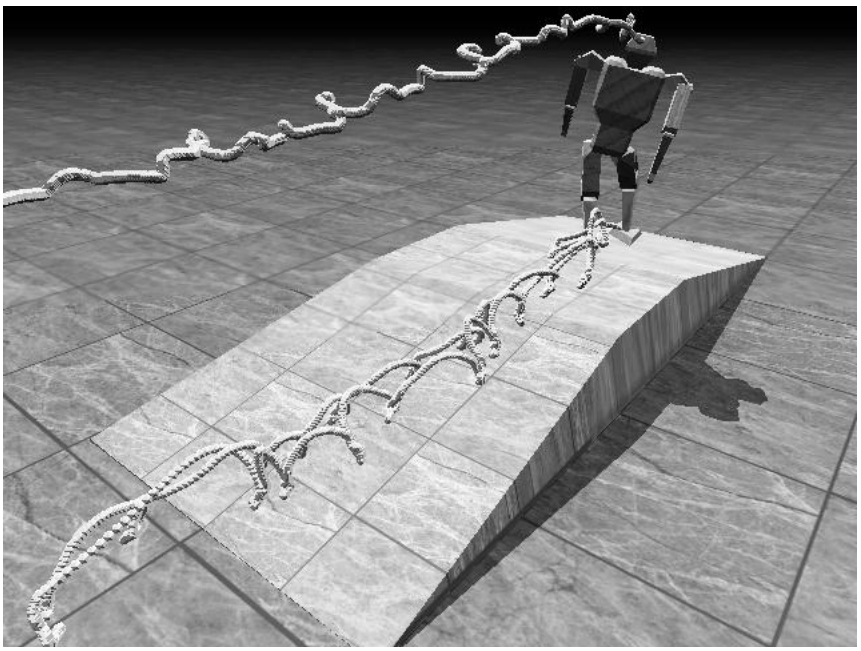


Figure 7.22: *Successful biped walking over the ramp, from another angle.*

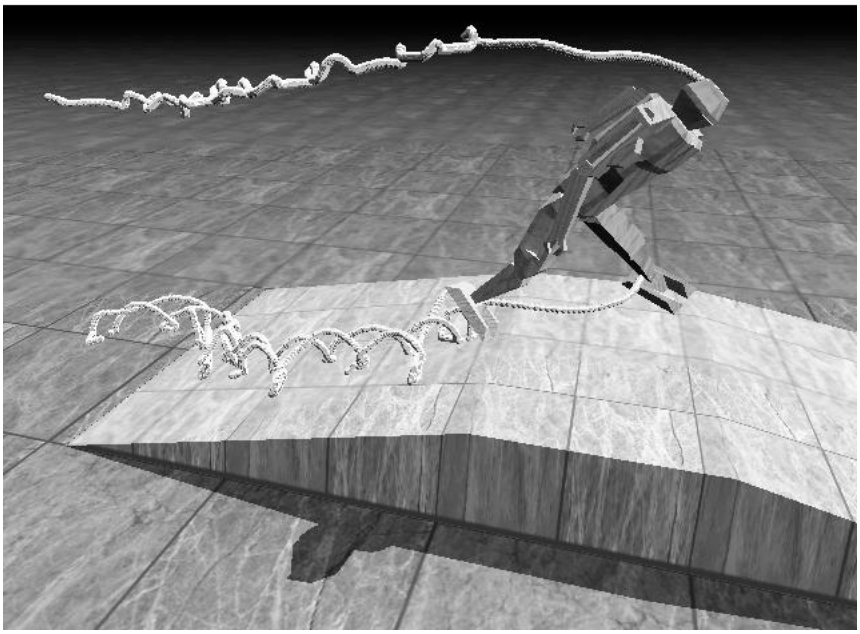


Figure 7.23: *Biped walking on slopes: during training the biped learns to walk on each gradient separately. Here it has successfully climbed the ramp but trips over at the top.*

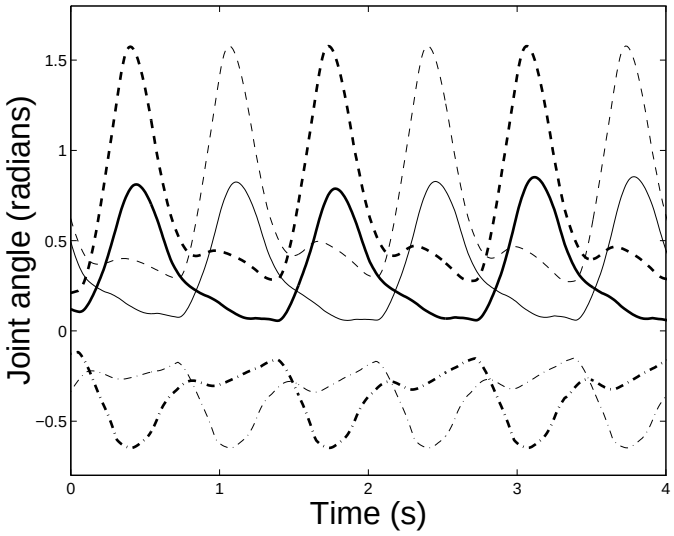


Figure 7.24: *Biped joint angles for three stepping cycles, before training. Key: left main hip joint (—), right main hip joint (**—**), left knee joint (– – –), right knee joint (**– – –**), left ankle joint (– · –), right ankle joint (**– · –**).*

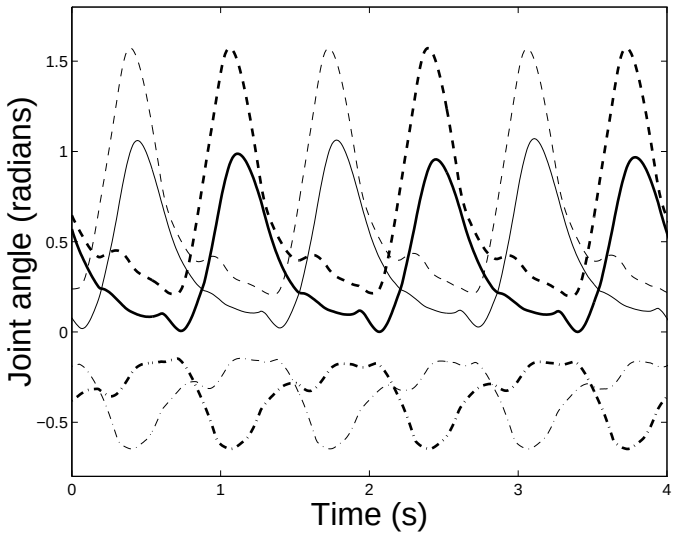


Figure 7.25: *Biped joint angles for three stepping cycles, after training. Key: left main hip joint (—), right main hip joint (▬), left knee joint (– – –), right knee joint (▬ ▬), left ankle joint (·····), right ankle joint (▬ · ▬).*

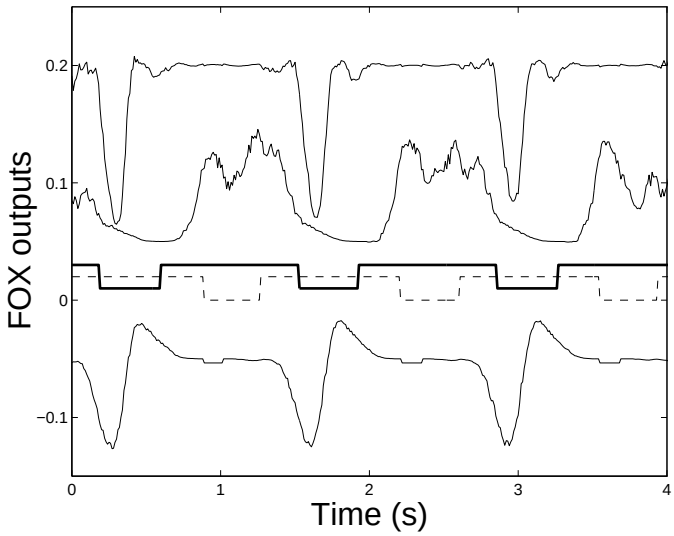


Figure 7.26: *FOX outputs for three stepping cycles, after training. Key: right leg FOX outputs (—), left foot sensor (– – –), right foot sensor (▬). The FOX outputs are, from top to bottom,* $f_2 + 0.2$, $f_4 + 0.05$, $f_1 - 0.05$.

7.8 Conclusion

The FOX controller can be used to successfully adjust control parameters to achieve stable walking in the biped robot. This is in spite of the fact that FOX is being used in a manner unanticipated by its theoretical formulation. There is no global error to be minimized, rather each FOX tries to improve some local error during part of the robot's motion. The eligibility profiles used by each FOX are derived from the designer's intuition (i.e. they are guessed), which is an acceptable procedure given the large amount of approximation that is allowed.

Once the biped was trained it had a similar performance to Laszlo's similarly configured biped [64], which used limit cycle control. But the system described here has the advantage that it can be trained on-line, and is thus far more suitable for implementation in a real robot.

The FOX error signals are not derived from a global error, so the biped's optimal behavior is not predefined. However, the biped's success can be judged in other ways: as it is trained it is able to walk for longer without falling over, it is able to walk more stably at the desired speed, and the body orientation deviates less from the desired orientation. Note that, as with the hopping robot, "perfect" control is never achieved because of the constraints imposed by the controller's internal structure.

The FOX modules are not given total control of the system, because the designer must constrain the robot's behavior via the controller design to whatever is appropriate for the environment. Because of this, perfect control is difficult to achieve, as not all the requirements can be satisfied simultaneously.

It is easy to assume that less care is needed when designing a FOX based controller, because FOX will be able to magically fix up any mistakes. But this is not really true: FOX modules make the system more adaptive, but care and attention is still needed to design a practical controller.

Chapter 8

Future work and conclusions

8.1 Suggestions for future work

There are several issues with regard to FOX which should be further investigated:

1. An algorithm for automatic synthesis of the A, B, and C matrices needs to be created. Such an algorithm would take as input a system's impulse response and a desired eligibility order. It would synthesize an eligibility model which is as accurate as possible within the trace precision requirements. A possible method is least squares minimization using a gradient search technique.

2. The most time consuming part of FOX controller design is the selection of the error functions and learning rates. Currently the learning rates are selected by trial and error. A more systematic method is desirable. Such a method would need information about the likely distribution of the error signal and the desired (approximate) magnitude of the FOX output. Methods to avoid overtraining should also be considered. Currently the best approach is to use overshoot training, as fewer parameters need to be guessed.

3. Other types of error function need to be investigated. The currently used ones (output limiting, overshoot etc) are useful but surely there are others.

4. As far as practical applications go, there obviously needs to be a lot more testing of FOX in real systems. More design experience is needed, and more design rules and guidelines should be formulated.

5. A FOX approximation theorem needs to be formalized. That is, exactly how much eligibility profile approximation can be tolerated for training to succeed? The current understanding of this issue is rather poor.

   An extension of this would be to show to what extent nonlinear systems can be controlled. Given a FOX approximation theorem, the way to proceed is to show that the nonlinear system local linear approximations are within the desired bounds in all useful regions of the FOX input space.

6. Currently FOX training must encompass all potentially useful regions of the CMAC input space. This is a limitation of the CMAC's local generalization, and it means that careful selection of the CMAC inputs and input parameters is required. FOX could be adapted to some more globally generalizing CMAC variant, for example a CMAC with adaptive input encoding. Also the properties

of multiple CMAC modules which each use a subset of the available inputs should be investigated (this is after all the way it works in the real cerebellum).

7. A better understanding of how to use one or more FOXs to service multiple constraints needs to be obtained. This is particularly true when the FOX eligibility profiles are approximations—how should they be chosen to prevent the independent error signals from simply having an additive effect on the FOX outputs?

8. FOX should be generalized to non-binary CMACs, and also to other neural network paradigms if this is possible.

9. The trace precision problem is a constant annoyance. Is there an implementation that avoids it?

10. FOX should be re-formulated for use with time-delay systems (this is a common class of system where the control signals are not manifested in the sensor readings until some minimum time has passed). Possibly this can be done by holding weight/eligibility values in "stasis" for the time delay period until their effect on the system is measurable, at which time they will be modified by the error signal. This would also be useful for modeling eligibility profiles with a significant initial zero portion.

11. A CMAC with a wrap-around (toroidial) weight table should be created as an alternative to using a two dimensional closed loop (circular) input trajectory as a means to avoid the local generalization discontinuity problem.

It has been demonstrated that FOX is a useful component in legged robot controllers, but obviously *much* more work needs to be done to explore the possibilities here. A much more comprehensive robot controller should be constructed that allows all aspects of movement to be learned, not just the handful of parameters for the single stereotyped walking motion studied here. In general a much better "brain design" approach is required. The existing methods using the DND language (see Appendix J) have problems when the system gets too large: many unforeseen interactions occur between system components, and the system also becomes hard to understand and visualize. Possibly a graphically based design approach needs to be used.

Finally, a robust biped brain needs to have far more built-in reflexes to compensate for a wider range of situations.

8.2 Conclusions

This thesis has developed the FOX adaptive controller from a theoretical consideration of the optimal control problem. FOX is based on the CMAC neural network, which is a simple model of the cerebellum, a part of the brain which helps control movement. FOX improves the cerebellar model by associating eligibility values with each CMAC weight. FOX is used in a configuration similar to the feedback-error (FBE) controller. With FBE an explicit reference trajectory must be generated to suit different situations. In contrast FOX will automatically find an error-minimizing trajectory that approaches a single reference path in a well defined way. It has also been shown that FOX can control some systems that FBE can not.

FOX implements a form of reinforcement learning which assigns each CMAC weight an eligibility value which controls how that weight is updated. Unlike other systems, FOX eligibilities can be vector quantities. The optimal eligibility update dynamics were shown to be the same as the system that is being controlled. Previous studies [68] have only provided loose guidelines for updating eligibilities.

A design methodology for using FOX in an adaptive control system was developed. The designer's expert knowledge about how a system should be controlled is split between an "internal" feedback controller added to the system and the choice of error signal (and error function) given to FOX. The system's impulse response is used to synthesize an eligibility profile, which determines how the eligibility values are updated. It was shown that a large amount of inaccuracy is allowable, so eligibility profile approximations can be made easily, although numerical precision issues limit the choices that can be made.

A highly efficient implementation was developed whose speed is independent of the number of CMAC weights and the controlled system parameters. This implementation overcomes the need to update each weight's eligibility at each time step. This is more sophisticated, faster and more useful than previous approaches such as [48].

FOX was successfully used in three real-time learning control system experiments, including the control of the classically difficult inverted pendulum and other nonlinear systems. It was discovered that FOX is easy to design with, and highly effective as long as training encompasses a large enough area of the state space. The most time consuming part of controller design was found to be the selection of error functions and learning parameters. Both feed-forward and feedback modes of control were investigated—feed-forward needs less CMAC inputs but requires known starting states, and feedback has more CMAC inputs but is more flexible when unanticipated states are encountered.

Finally, FOX was shown to be a useful learning component in two autonomous legged robot controllers—a simulated hopping monoped and walking biped. It was used to learn a handful of parameters and reference adjustments for hard-wired controllers, to coordinate their existing behavior. A cunning use of FOX gates, training signals and error functions was necessary. It was found that with a number of FOX modules in a system, a badly trained one could be disguised by all the well trained ones, so careful checking was required.

Successful walking was implemented in the biped robot. Despite the seeming sophistication of this walking behavior, the biped controller is not robust enough for real world walking because of its simplicity. The problem of controlling a biped which must survive in an unstructured environment may be too hard for the relatively simple design techniques presented here.

Finally, a lot of further work needs to be done to make FOX a practical industrial adaptive controller. FOX can be part of the toolbox for autonomous robot design, but of course many other techniques will be necessary too. Most of the techniques that will be useful in future designs have not even been discovered yet!

Appendix A

Definitions and basic concepts

A.1 Notation

The following mathematical notation conventions will be used:

- All vectors are column vectors unless otherwise stated, so the dot product $\mathbf{x} \bullet \mathbf{y}$ is equal to $\mathbf{x}^T\mathbf{y}$.
- $\mathbf{x}^{(i)}$ is the i'th element of vector $\mathbf{x}$.
- Vector derivatives: for vectors $\mathbf{x}$ (of size $n \times 1$) and $\mathbf{y}$ (of size $m \times 1$), the quantity $\mathrm{d}\mathbf{x}/\mathrm{d}\mathbf{y}$ is a matrix of size $n \times m$:

$$\frac{\mathrm{d}\,\mathbf{x}}{\mathrm{d}\,\mathbf{y}} = \begin{bmatrix} \frac{\mathrm{d}\,\mathbf{x}^{(1)}}{\mathrm{d}\,\mathbf{y}^{(1)}} & \cdots & \frac{\mathrm{d}\,\mathbf{x}^{(1)}}{\mathrm{d}\,\mathbf{y}^{(m)}} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}\,\mathbf{x}^{(n)}}{\mathrm{d}\,\mathbf{y}^{(1)}} & \cdots & \frac{\mathrm{d}\,\mathbf{x}^{(n)}}{\mathrm{d}\,\mathbf{y}^{(m)}} \end{bmatrix} \tag{A.1}$$

  Similarly for $\partial\mathbf{x}/\partial\mathbf{y}$.

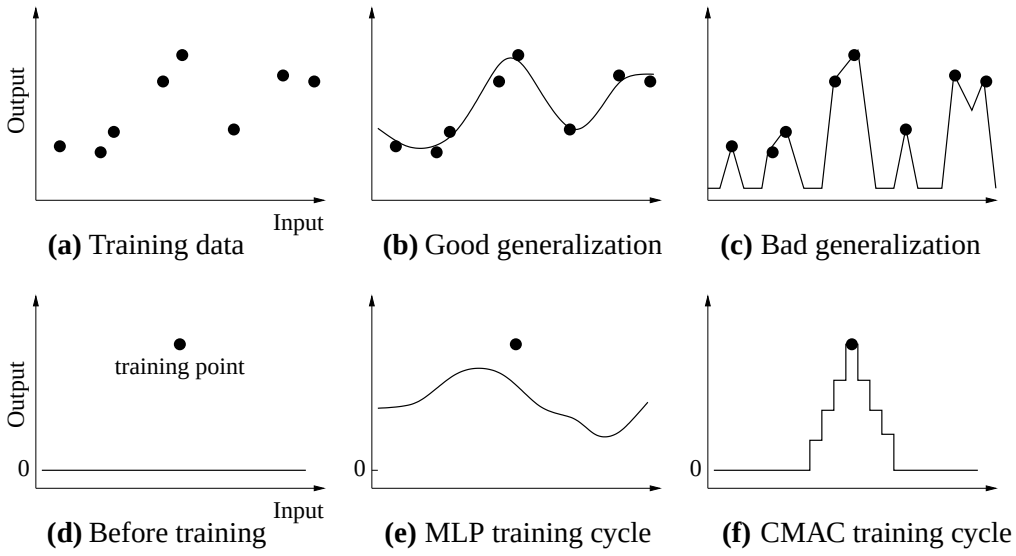
A.2 Parameterized mappings and generalization

A parameterized mapping is a function of the following form:

$$\mathbf{x} = f(\mathbf{y}, \mathbf{w}) \tag{A.2}$$

where $\mathbf{y}$ is the $n_y \times 1$ input vector, $\mathbf{x}$ is the $n_x \times 1$ output vector, and $\mathbf{w}$ is the $n_w \times 1$ vector of parameters or *weights*. If the function f is a *general approximator* then the weights $\mathbf{w}$ can be selected so that f will approximate any desired $\mathbf{y} \rightarrow \mathbf{x}$ mapping, within some constraints. In other words the mapping is flexible. There are as many different types of general approximator as there are researchers in the field. Many popular approaches are based on neural networks and fuzzy logic, including multi-layer perceptrons, radial basis function networks, and fuzzy neural networks [49].

It is sometimes useful to talk about trajectories in input space and output space. The *input space* is the n_y dimensional space in which a single input $\mathbf{y}$ is represented as a point. An input space trajectory is a continuous one dimensional curve in this space, which represents an input that changes with time. A similar convention applies to the output space.



(a) Training data **(b)** Good generalization **(c)** Bad generalization

(d) Before training **(e)** MLP training cycle **(f)** CMAC training cycle

Figure A.1: *What generalization means.*

A.3 Some notes on training and generalization

Training is the process of finding a weight vector $\mathbf{w}$ that gives a desired mapping. It is common to train general approximators by presenting *training points* to a training algorithm. Each training point is a pair $(\mathbf{y}_i, \mathbf{x}_i^d)$ where $\mathbf{y}_i$ is an input and $\mathbf{x}_i^d$ is the corresponding desired output (i indexes the training point number). This is called *target training*. A one dimensional example is shown in figure A.1a. An ideal approximator, once trained, will generalize the training points to the surrounding space, producing a mapping which is consistent with the training data and which ignores noise while representing any underlying structural features (figure A.1b). A bad approximator may fit the training data well without correctly interpolating the in-between areas (figure A.1c).

Some approximators such as the multi-layer perceptron (MLP, Appendix C) perform *global generalization*. Every weight has an influence on the mapping over the entire input space. This means that when each data point is presented to the training algorithm, the entire mapping is adjusted. While the mapping at the training point becomes better, the error at other points can be made worse. However the process usually converges to a mapping which accommodates all the data points. Figure A.1d shows one data point and figure A.1e shows a possible result of one MLP training iteration on that point.

Other approximators (such as the CMAC neural network, Chapter 3) perform *local generalization*. Every weight only influences a small region of the entire input space. This means that when each data point is presented to the training algorithm, only a small region of the mapping is adjusted (figure A.1f). Approximators that exhibit local generalization are often quicker to train because the correct value for each weight depends on fewer training points. However, they often give a poorer approximation to the function from which the training data is drawn. One reason for this is that they can easily over-fit the training data. Another is that they may be unable to correctly interpolate the training data in the regions of the input space where there are few data points.

Another common training method is called *error training*. The training data consists of pairs $(\mathbf{y}_i, \Delta\mathbf{x}_i)$ where $\mathbf{y}_i$ is an input and $\Delta\mathbf{x}_i$ is the desired change in output, called the output error. The training algorithm adjusts the mapping so that the output $\mathbf{x}_i$ for the given $\mathbf{y}_i$ becomes $\mathbf{x}_i + \alpha\,\Delta\mathbf{x}_i$, i.e. so it moves in

the *direction* $\Delta\mathbf{x}_i$ (α is an arbitrary number). Note that target training is a special case of error training where $\Delta\mathbf{x}_i = \mathbf{x}_i^d - \mathbf{x}_i$ and $\alpha = 1$.

A common method for target training is gradient descent. A scalar error quantity E is defined like this:

$$E = \sum_i \Big(\mathbf{x}_i^d - \mathbf{x}_i\Big)^2 \tag{A.3}$$

$$= \sum_i \Big(\mathbf{x}_i^d - f(\mathbf{y}_i, \mathbf{w})\Big)^2 \tag{A.4}$$

When all the outputs match the *desired* outputs, the error E is minimized. To minimize E the weight vector $\mathbf{w}$ is moved along the negative gradient $\mathrm{d}E/\mathrm{d}\mathbf{w}$ in small increments. In other words, the following is done at each step:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \left[\frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{w}}\right]^T \tag{A.5}$$

$$\leftarrow \mathbf{w} - \alpha \left[\frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{x}} \cdot \frac{\mathrm{d}\,\mathbf{x}}{\mathrm{d}\,\mathbf{w}}\right]^T \tag{A.6}$$

$$\leftarrow \mathbf{w} + 2\alpha \left[\frac{\mathrm{d}\,\mathbf{x}}{\mathrm{d}\,\mathbf{w}}\right]^T \cdot \left(\sum_i \mathbf{x}_i^d - \mathbf{x}_i\right) \tag{A.7}$$

The parameter α is the learning rate. Gradient descent can also be applied to *error training* if we define

$$E = \sum_i (\Delta\mathbf{x}_i)^2 \tag{A.8}$$

In which case the training algorithm becomes

$$\mathbf{w} \leftarrow \mathbf{w} + 2\alpha \left[\frac{\mathrm{d}\,\mathbf{x}}{\mathrm{d}\,\mathbf{w}}\right]^T \cdot \left(\sum_i \Delta\mathbf{x}_i\right) \tag{A.9}$$

Often it is computationally useful to perform training using an approximation of the gradient vector $\mathrm{d}E/\mathrm{d}\mathbf{w}$. In such cases it is useful to know under what circumstances the approximation is valid. That obviously depends on the specifics of the approximation, but there are some general observations to be made. Consider figure A.2, which shows how the training algorithm operates in weight-error space. The best error reduction is obviously obtained by traveling along the vector $\mathrm{d}E/\mathrm{d}\mathbf{w}$. However, the error will still be reduced (at least in the short term) along any path within $90°$ of this vector. Such paths make up exactly half of all the possible directions that the weight vector could travel in. So, in an extremely loose sense, it could be said that half of all approximations will work (i.e. will reduce the error). Of course the complicated geometry of the weight-error space means that some directions (i.e. some approximation) will be more useful than others.

One final point: general approximators are often used as components of dynamic systems, that is systems that operate over time. To analyze such systems the concept of *unfolding* is extremely useful (figure A.3). Each time step is represented separately and the system's operation over all time is treated as a whole, where signals propagate from start to finish in a single instant.

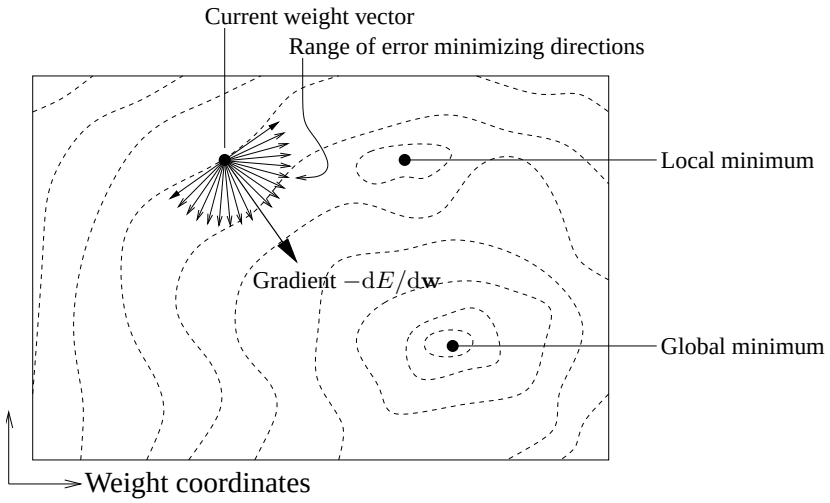


Figure A.2: *Gradient descent in weight-error space.*

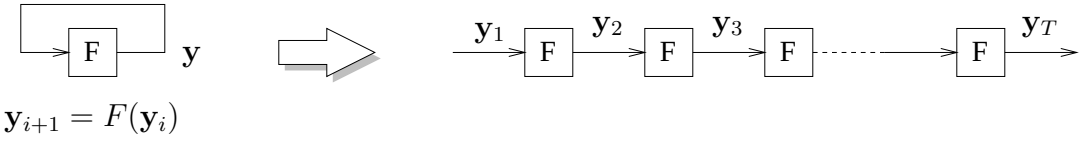


Figure A.3: *Unfolding a dynamic system.*

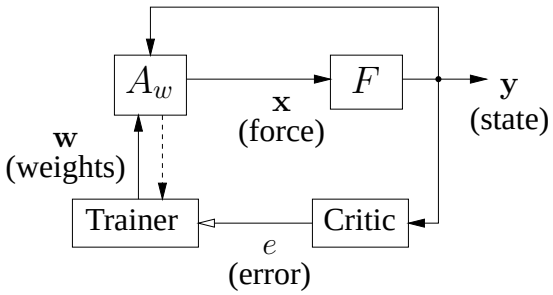


Figure A.4: *A basic adaptive control system.*

A.4 Adaptive control

Figure A.4 shows the form of a simple adaptive control system. F is the process being controlled, which obeys the dynamical equation:

$$\frac{\mathrm{d}\,\mathbf{y}}{\mathrm{d}\,t} = F(\mathbf{y}, \mathbf{x}) \tag{A.10}$$

The vector $\mathbf{y}$ is the system state (for a mechanical system this is typically positions and velocities). It is fed into a stateless controller A_w to produce the vector $\mathbf{x}$, which is the process control force. The function A_w is some kind of "black box" (like a general approximator) that makes decisions about how to control the system. It is parameterized by the weight vector $\mathbf{w}$.

If there is some detailed knowledge of how the controller should work then it may be possible to choose a specific form for the function A and the parameters w that will allow optimal control and make the adaptation problem easier. This can result in efficient control and adaptation when the system F is simple or linear, and its dynamics are well known.

The alternative is to choose A_w to be a general approximator. This means that A_w should be flexible enough that the weights w can be adjusted to achieve *whatever* mapping will eventually be required. In practice it is not possible to specify every possible function with a finite set of parameters, so A_w is chosen such that it can merely *approximate* a wide set of useful functions. This is a useful approach if the required A_w is unknown, for example because the system is nonlinear or has unknown dynamics, or perhaps because the control engineer just can't be bothered to work it out.

The goal of adaptive control is to adjust the parameters $\mathbf{w}$ so that A_w provides the best control (the definition for "best" depends on exactly what you are trying to do). In *optimal* control an error quantity E is defined and the weights $\mathbf{w}$ are adjusted to minimize it. The training process usually involves a *critic* which determines an error signal e, and a *trainer* which uses the error information to adjust the weights (using gradient descent for example).

In figure A.4 it is assumed that the entire system state vector can be measured by the controller. In reality only part of the state may be measurable. However, measurability issues will be mostly ignored in this thesis: it will be assumed that enough of the state vector can be determined to provide adequate control.

Appendix B

Chain-rule propagation algorithms

B.1 Introduction[1]

Adaptive dynamic systems (dynamic systems which learn to change over time) may be implemented using neural networks as the adaptive component. Such systems include adaptive process controllers [86], adaptive filters, and multi-layer networks with output feedback connections [130]. This appendix outlines some of the "traditional" gradient based training approaches for these systems, to contrast with the other methods presented in this thesis.

In the neural network literature, training algorithms for such systems are generally of two types: those which propagate derivative information forwards in time, and those which propagate it backwards. These two types of algorithm are derived and analyzed for a simple prototype system. It is shown that they are very closely related because they compute the same components of the gradient vector but in a different order. The well known computational properties of each algorithm are then explained using a simple matrix multiplication analogy. Extensions of the prototype to control systems are demonstrated.

A discrete-time prototype of such systems is shown in figure B.1. At time t in this figure, F_t is the "system function" (which incorporates the adaptive component), the vector $\mathbf{y}_t$ contains the system state variables, and the scalar c_t is the time step cost. The vector $\mathbf{w}$ contains the parameters (or "weights") that are fed to the adaptive components of all F_t (the vectors $\mathbf{w}_t$ are all equal to $\mathbf{w}$). The goal of training

[1] This appendix is derived from the author's paper in the Proceedings of the 1995 ANNES conference [114].

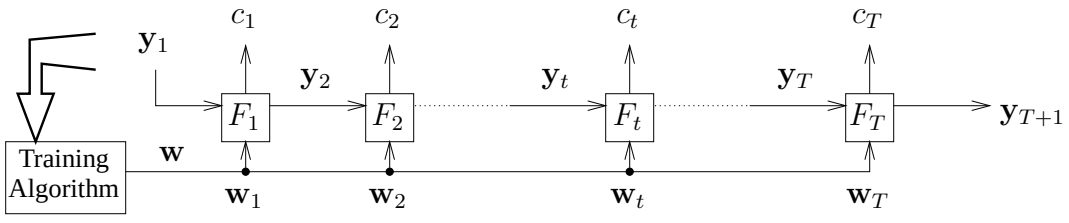


Figure B.1: *A generic discrete-time adaptive dynamic system.*

(or adaption) is to adjust the weights $\mathbf{w}$ to minimize

$$E = \sum_{i=1}^{T} c_i \tag{B.1}$$

This prototype can represent many different learning problems by selecting the F_t appropriately. For example, F_t could represent a process and its controller at time t, so that by minimizing E we ensure optimal process control. This is elaborated further in section B.4.

To train such systems using gradient based methods[2] we must calculate the gradient of the weight vector with respect to E. Two commonly cited algorithms for computing this are backpropagation-through-time (BPTT) and real-time-backpropagation (RTBP). BPTT [96, 95] finds the system states forward in time and then propagates information backwards through time to find the gradient. RTBP [130, 99] allows the gradient to be computed forward in time as the system states become known. RTBP is known to be more computationally expensive than BPTT but it does not require a backwards-though-time phase, so it can be performed on-line[3].

These two algorithms are equivalent in terms of what they compute, but because few authors derive both for the same system (and because notation differs between authors) it is not always obvious that they are interchangeable.

This appendix shows how both types of learning algorithm can be derived for the prototype system in figure B.1. In both derivations, application of the chain rule shows that information must be propagated between adjacent time steps for the gradient to be computed. This can be done either forwards or backwards in time. Thus these algorithms are named forward propagation (FP) and backward propagation (BP).

Concepts similar to [12] are presented below, though in that paper RTBP and BPTT were related using the inter-reciprocity of signal flow graphs. Here the application of matrix chain-rule techniques are emphasized.

B.2 Derivation of FP and BP

B.2.1 Notation

Notational standards for writing vector derivatives vary widely between authors, so we must define our own here. We define the derivative of vector $\mathbf{a}$ (of size n_a) with respect to vector $\mathbf{b}$ (of size n_b) as the matrix $\mathrm{d}\mathbf{a}/\mathrm{d}\mathbf{b}$ (of size $n_a \times n_b$), whose (i,j)th element is $\mathrm{d}a_i/\mathrm{d}b_j$ (a_i and b_i are elements of $\mathbf{a}$ and $\mathbf{b}$). The partial derivative matrix $\partial\mathbf{a}/\partial\mathbf{b}$ is defined similarly. Note the following convention for partial derivatives: $\partial\mathbf{a}/\partial\mathbf{b}$ is calculated assuming that only $\mathbf{b}$ varies while all other quantities in the *definition* of $\mathbf{a}$ are held constant. The total derivative $\mathrm{d}\mathbf{a}/\mathrm{d}\mathbf{b}$ assumes that *all* of $\mathbf{b}$'s influences on $\mathbf{a}$ are accounted for. For example, if

$$\mathbf{a} \triangleq F_1(\mathbf{b}, \mathbf{c}), \quad \mathbf{c} \triangleq F_2(\mathbf{b}), \qquad \left(\begin{array}{c} \mathbf{b} \\ \rightarrow \boxed{F_1} \rightarrow \mathbf{a} \\ \rightarrow \boxed{F_2} \rightarrow \mathbf{c} \end{array} \right) \tag{B.2}$$

[2] Of which gradient descent is the simplest.

[3] An on-line training algorithm is one in which the training process occurs during the operation of the system. On-line algorithms tend to be simpler to implement.

then

$$\frac{\mathrm{d}\,\mathbf{a}}{\mathrm{d}\,\mathbf{b}} = \frac{\partial\,\mathbf{a}}{\partial\,\mathbf{b}} + \frac{\partial\,\mathbf{a}}{\partial\,\mathbf{c}} \cdot \frac{\mathrm{d}\,\mathbf{c}}{\mathrm{d}\,\mathbf{b}} \tag{B.3}$$

It can be easily shown that the chain rule holds in the vectorial case, with one *caveat*: it must be remembered that matrix derivative quantities are not commutative, so that

$$\frac{\partial\,\mathbf{a}}{\partial\,\mathbf{c}} \cdot \frac{\mathrm{d}\,\mathbf{c}}{\mathrm{d}\,\mathbf{b}} \neq \frac{\mathrm{d}\,\mathbf{c}}{\mathrm{d}\,\mathbf{b}} \cdot \frac{\partial\,\mathbf{a}}{\partial\,\mathbf{c}} \tag{B.4}$$

B.2.2 Forward propagation algorithm (FP)

The FP algorithm is now derived. Using the chain rule and the definition of E we get:

$$\frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{w}} = \sum_{i=1}^{T} \frac{\partial\,E}{\partial\,c_i} \cdot \frac{\mathrm{d}\,c_i}{\mathrm{d}\,\mathbf{w}} \tag{B.5}$$

$$= \sum_{i=1}^{T} \left(\frac{\partial\,c_i}{\partial\,\mathbf{w}} + \frac{\partial\,c_i}{\partial\,\mathbf{y}_i} \cdot \frac{\mathrm{d}\,\mathbf{y}_i}{\mathrm{d}\,\mathbf{w}} \right) \tag{B.6}$$

(Note that $\partial E/\partial c_i = 1$). If the values of $\mathbf{y}_i$ and $\mathbf{w}$ are known (i.e. the system has reached at least time step i) then the quantities $\partial c_i/\partial \mathbf{w}$ and $\partial c_i/\partial \mathbf{y}_i$ can be computed without any further information — assuming of course that our knowledge of the function F_i is sufficient. We can find $\mathrm{d}\mathbf{y}_i/\mathrm{d}\mathbf{w}$ if we know the previous $\mathrm{d}\mathbf{y}_{i-1}/\mathrm{d}\mathbf{w}$, using the chain rule:

$$\frac{\mathrm{d}\,\mathbf{y}_i}{\mathrm{d}\,\mathbf{w}} = \frac{\partial\,\mathbf{y}_i}{\partial\,\mathbf{w}} + \frac{\partial\,\mathbf{y}_i}{\partial\,\mathbf{y}_{i-1}} \cdot \frac{\mathrm{d}\,\mathbf{y}_{i-1}}{\mathrm{d}\,\mathbf{w}} \tag{B.7}$$

Note that $\mathrm{d}\mathbf{y}_1/\mathrm{d}\mathbf{w} = 0$. Thus the FP algorithm is:

- Set $\mathrm{d}\mathbf{y}_1/\mathrm{d}\mathbf{w} = 0$ and $\mathrm{d}E/\mathrm{d}\mathbf{w} = 0$
- For $t = 1 \ldots T$ do
  - Calculate c_t and $\mathbf{y}_{t+1}$ using F_t (or measure these things, depending on the system).
  - Calculate $\mathrm{d}c_i/\mathrm{d}\mathbf{w}$ and add it to $\mathrm{d}E/\mathrm{d}\mathbf{w}$ (using equations B.5–B.6).
  - Calculate $\mathrm{d}\mathbf{y}_{t+1}/\mathrm{d}\mathbf{w}$ from the value of $\mathrm{d}\mathbf{y}_t/\mathrm{d}\mathbf{w}$ (using equation B.7).

B.2.3 Backward propagation algorithm (BP)

The BP algorithm is now derived. This time $\mathrm{d}E/\mathrm{d}\mathbf{w}$ is split up into different components from the FP approach:

$$\frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{w}} = \sum_{i=1}^{T} \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{w}_i} \tag{B.8}$$

$$= \sum_{i=1}^{T} \left(\frac{\partial\,E}{\partial\,c_i} \cdot \frac{\partial\,c_i}{\partial\,\mathbf{w}} + \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_{i+1}} \cdot \frac{\partial\,\mathbf{y}_{i+1}}{\partial\,\mathbf{w}} \right) \tag{B.9}$$

$$= \sum_{i=1}^{T} \left(\frac{\partial\,c_i}{\partial\,\mathbf{w}} + \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_{i+1}} \cdot \frac{\partial\,\mathbf{y}_{i+1}}{\partial\,\mathbf{w}} \right) \tag{B.10}$$

Method	*Time taken*	*Storage required*
FP	$\mathcal{O}\Big(T\, n_y^2\, n_w\Big)$	$\mathcal{O}\Big(n_y\, n_w\Big)$
BP	$\mathcal{O}\Big(T(n_y^2 + n_y\, n_w)\Big)$	$\mathcal{O}\Big(T\, n_y\Big)$

Table B.1: *Efficiency of the FP and BP algorithms.*

Note that the derivative $\mathrm{d}E/\mathrm{d}\mathbf{w}_i$ assumes that only the weight vector into function F_i is varied while all others are held constant. The quantity $\mathrm{d}E/\mathrm{d}\mathbf{y}_i$ can be calculated from the subsequent $\mathrm{d}E/\mathrm{d}\mathbf{y}_{i+1}$, because if $\mathbf{y}_i$ is varied then only subsequent $\mathbf{y}$ and c values will change. Thus, using the chain rule again:

$$\frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_i} = \frac{\partial\, c_i}{\partial\, \mathbf{y}_i} + \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_{i+1}} \cdot \frac{\partial\, \mathbf{y}_{i+1}}{\partial\, \mathbf{y}_i} \tag{B.11}$$

Note that $\mathrm{d}E/\mathrm{d}\mathbf{y}_{T+1} = 0$. Thus the BP algorithm is:

- For $t = 1 \ldots T$ do
  - Calculate (or measure) $\mathbf{y}_{t+1}$ and c_t, using F_t.
- Set $\mathrm{d}E/\mathrm{d}\mathbf{w} = 0$ and $\mathrm{d}E/\mathrm{d}\mathbf{y}_{T+1} = 0$
- For $t = T \ldots 1$ do
  - Calculate $\mathrm{d}E/\mathrm{d}\mathbf{y}_t$ from $\mathrm{d}E/\mathrm{d}\mathbf{y}_{t+1}$ (using equation B.11).
  - Calculate $\mathrm{d}E/\mathrm{d}\mathbf{w}_t$ and add it to $\mathrm{d}E/\mathrm{d}\mathbf{w}$ (using equations B.8–B.10).

B.3 Efficiency of FP and BP

B.3.1 Space and time requirements

Table B.1 shows the order[4] of the time and storage space requirements for the two algorithms. Here n_y and n_w are the sizes of the $\mathbf{y}_t$ and $\mathbf{w}$ vectors respectively. The time values were determined assuming that the "local" partial derivatives of each system function F_t (i.e. $\partial\mathbf{y}_{t+1}/\partial\mathbf{y}_t$, $\partial\mathbf{y}_{t+1}/\partial\mathbf{w}$, $\partial c_t/\partial\mathbf{y}_t$, $\partial c_t/\partial\mathbf{w}$) are relatively fast to compute. Thus in both cases the time taken is influenced most by the matrix multiplications required by the chain rule.

The storage space values correspond to the amount of information which needs to be recorded about the system for later use. In the FP algorithm this is the matrix $\mathrm{d}\mathbf{y}_t/\mathrm{d}\mathbf{w}$, in the BP algorithm this is the system state $\mathbf{y}_t$ stored over all time.

FP is slower than BP by an approximate factor of

$$\eta_{\text{time}} \approx \frac{1}{n_w} + \frac{1}{n_y} \tag{B.12}$$

For a large system[5] this means that FP can be significantly slower than BP. The storage space required by FP is less than that of BP by a factor of

$$\eta_{\text{space}} \approx \frac{T}{n_w} \tag{B.13}$$

[4]The "Big-Oh" notation used in this table just shows the dominant terms in the time and space expressions, for large systems.
[5]i.e. where n_y and n_w are large

When $T \gg n_w$ (for example when the system is being run for a long time with a small time step) FP uses much less storage space than BP.

The advantage of BP is that it is fast. The advantages of FP are that it can use less storage space than BP, and it can be implemented as an on-line algorithm (where learning occurs forward in time as the system operates).

B.3.2 Why is FP slower than BP?

It has been shown above that FP is slower than BP, but the derivations of the algorithms do not make the fundamental reason for this very clear.

Consider the simplified situation[6] where all the $c_i = 0$ except for c_T. A closed form expression for $\mathrm{d}E/\mathrm{d}\mathbf{w}$ can be found by "unfolding" the BP algorithm over time. The result can be expressed in terms of the local partial derivatives of each system block F_t, as follows:

$$\begin{aligned} \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_i} &= \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_{i+1}}\,\mathbf{Q}_i && \text{(B.14)} \\ &= \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_T}\Big[\mathbf{Q}_{T-1}\mathbf{Q}_{T-2}\cdots\mathbf{Q}_{i+1}\mathbf{Q}_i\Big] && \text{(B.15)} \\ &= \frac{\partial\,c_T}{\partial\,\mathbf{y}_T}\Big[\mathbf{Q}_{T-1}\mathbf{Q}_{T-2}\cdots\mathbf{Q}_{i+1}\mathbf{Q}_i\Big] && \text{(B.16)} \end{aligned}$$

where $\mathbf{Q}_i = \partial\mathbf{y}_{i+1}/\partial\mathbf{y}_i$, and

$$\begin{aligned} \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{w}} &= \frac{\partial\,c_T}{\partial\,\mathbf{w}} + \sum_{i=1}^{T}\frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_{i+1}}\,\mathbf{P}_{i+1} && \text{(B.17)} \\ &= \frac{\partial\,c_T}{\partial\,\mathbf{w}} + \frac{\partial\,c_T}{\partial\,\mathbf{y}_T}\Big[\mathbf{Q}_{T-1}\cdots\mathbf{Q}_3\mathbf{Q}_2\mathbf{P}_2 + \mathbf{Q}_{T-1}\cdots\mathbf{Q}_4\mathbf{Q}_3\mathbf{P}_3 + \cdots\cdots \\ &\qquad +\mathbf{Q}_{T-1}\mathbf{P}_{T-1} + \mathbf{P}_T\Big] && \text{(B.18)} \end{aligned}$$

where $\mathbf{P}_i = \partial\mathbf{y}_i/\partial\mathbf{w}$. The FP algorithm can also be unfolded in this way to yield the same expression. The difference between the two algorithms is the order in which the terms of the form

$$\frac{\partial\,c_T}{\partial\,\mathbf{y}_T}\mathbf{Q}_{T-1}\cdots\mathbf{Q}_{i+1}\mathbf{Q}_i\mathbf{P}_i \qquad \text{(B.19)}$$

are constructed. In the FP algorithm these terms are built up right-to-left because the values earliest in time are on the right. Thus the intermediate products have the form

$$\mathbf{Q}_k\mathbf{Q}_{k-1}\cdots\mathbf{Q}_{i+1}\mathbf{Q}_i\mathbf{P}_i \qquad (k > i) \qquad \text{(B.20)}$$

where k is the "iteration step". This quantity is a $n_y \times n_w$ matrix, so pre-multiplying it by $\mathbf{Q}_{k+1}$ requires $\mathcal{O}(n_y^2\,n_w)$ time. In the BP algorithm these terms are built up left-to-right, so the intermediate products have the form

$$\frac{\partial\,c_T}{\partial\,\mathbf{y}_T}\mathbf{Q}_{T-1}\cdots\mathbf{Q}_{k+1}\mathbf{Q}_k \qquad (k < T) \qquad \text{(B.21)}$$

[6] This simplification is made to keep the equations manageable—it does not affect the result.

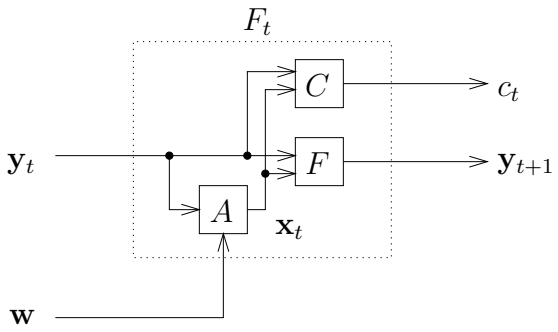


Figure B.2: *A system function for a stateless controller.*

FP

$$\frac{\mathrm{d}\,\mathbf{y}_{i+1}}{\mathrm{d}\,\mathbf{w}} = \left(\frac{\partial\,\mathbf{y}_{i+1}}{\partial\,\mathbf{y}_i} + \frac{\partial\,\mathbf{y}_{i+1}}{\partial\,\mathbf{x}_i}\cdot\frac{\partial\,\mathbf{x}_i}{\partial\,\mathbf{y}_i}\right)\cdot\frac{\mathrm{d}\,\mathbf{y}_i}{\mathrm{d}\,\mathbf{w}} + \frac{\partial\,\mathbf{y}_{i+1}}{\partial\,\mathbf{x}_i}\cdot\frac{\partial\,\mathbf{x}_i}{\partial\,\mathbf{w}}$$

$$\frac{\mathrm{d}\,c_i}{\mathrm{d}\,\mathbf{w}} = \left(\frac{\partial\,c_i}{\partial\,\mathbf{x}_i}\cdot\frac{\partial\,\mathbf{x}_i}{\partial\,\mathbf{y}_i} + \frac{\partial\,c_i}{\partial\,\mathbf{y}_i}\right)\frac{\mathrm{d}\,\mathbf{y}_i}{\mathrm{d}\,\mathbf{w}} + \frac{\partial\,c_i}{\partial\,\mathbf{x}_i}\cdot\frac{\partial\,\mathbf{x}_i}{\partial\,\mathbf{w}}$$

BP

$$\frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_i} = \frac{\partial\,c_i}{\partial\,\mathbf{y}_i} + \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_{i+1}}\left(\frac{\mathrm{d}\,\mathbf{y}_{i+1}}{\mathrm{d}\,\mathbf{y}_i} + \frac{\partial\,\mathbf{y}_{i+1}}{\partial\,\mathbf{x}_i}\cdot\frac{\partial\,\mathbf{x}_i}{\partial\,\mathbf{y}_i}\right)$$

$$\frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{w}_i} = \left(\frac{\partial\,c_i}{\partial\,\mathbf{x}_i} + \frac{\mathrm{d}\,E}{\mathrm{d}\,\mathbf{y}_{i+1}}\cdot\frac{\partial\,\mathbf{y}_{i+1}}{\partial\,\mathbf{x}_i}\right)\frac{\partial\,\mathbf{x}_i}{\partial\,\mathbf{w}}$$

Table B.2: *FP and BP propagation equations for a control system.*

This quantity is a $1 \times n_y$ matrix, so post-multiplying it by $\mathbf{Q}_{k-1}$ requires $\mathcal{O}(n_y^2)$ time, and the final post-multiplication by $\mathbf{P}_i$ requires $\mathcal{O}(n_y\, n_w)$ time.

Thus FP is slower than BP because a much larger amount of information needs to be propagated from step to step (FP and BP propagate $n_y \times n_w$ and $1 \times n_y$ matrices respectively between time steps). This happens (in the above example) because in FP the value of $\partial c_T/\partial \mathbf{y}_T$ is not known until the end of the training process and so a large amount of information must be carried through time to allow for its unknown value.

B.4 Control system extension to the prototype

To solve a particular learning problem, the functions F_t in the prototype must be specified in more detail. Figure B.2 shows an example of this for a stateless controller. Here F is the discrete transfer function of the process or plant, and A is the adaptive controller which produces the control output $\mathbf{x}_t$. A is adapted via the weight vector $\mathbf{w}$. The cost function C can be designed so that the minimum total cost corresponds to optimal system control in some sense.

It is relatively easy to derive the propagation equations to use in the FP and BP algorithms. For

reference these equations are shown in table B.2. In these equations, terms of the form

$$\frac{\partial\, k}{\partial\, \mathbf{x}_i} \cdot \frac{\partial\, \mathbf{x}_i}{\partial\, \mathbf{y}_i} \qquad \text{and} \qquad \frac{\partial\, k}{\partial\, \mathbf{x}_i} \cdot \frac{\partial\, \mathbf{x}_i}{\partial\, \mathbf{w}}$$

(where k is some vector or scalar) imply backpropagation operations in the controller A, i.e. we are trying to find an input gradient ($\partial k/\partial \mathbf{y}_i$ or $\partial k/\partial \mathbf{w}$) given an output gradient ($\partial k/\partial \mathbf{x}_i$). If A is a feed-forward neural network (such as a multi-layer perceptron) then these backpropagation operations can be computed using the standard neural network backpropagation equations. Note that doing this is usually faster (and uses less storage) than computing all the elements of $\partial \mathbf{x}_i/\partial \mathbf{y}_i$ or $\partial \mathbf{x}_i/\partial \mathbf{w}$ and performing the matrix multiplication.

Appendix C

The multi-layer perceptron

The multi-layer perceptron (MLP) neural network shown in figure C.1 is the most typical of the models used in current neural networks research. It is the work-horse of many practical systems. The MLP architecture will be described here and its backpropagation training algorithm will be derived.

The network is made up of several layers of artificial neurons. Each neuron is connected to every neuron in the immediately adjacent layers. The following notation is used:

L	The number of layers.
N_ℓ	The number of neurons in layer ℓ.
P	The number of training patterns.
u^p_{1j}	The value of input neuron j (in layer 1) for pattern p ($j = 1 \cdots N_1$, $p = 1 \cdots P$)
$u^p_{\ell j}$	The output of neuron j in layer ℓ for pattern p ($j = 1 \cdots N_\ell$, $\ell = 1 \cdots L$).
$a^p_{\ell j}$	The "activation" of neuron j in layer ℓ for pattern p ($\ell = 2 \cdots L$).
$w_{\ell ji}$	The weight connecting neuron i in layer $\ell - 1$ to neuron j in layer ℓ.
$\theta_{\ell j}$	The "threshold" for neuron j in layer ℓ.
d^p_j	The desired output (the desired value of a^p_{Lj}) for training pattern p.
E_p	The sum squared error (at the outputs) for training pattern p.
E	The sum squared error over all training patterns.

The inputs are u^p_{1j} and the outputs are a^p_{Lj}. Each neuron of the MLP (in the hidden and output layers) does the following:

$$u^p_{\ell j} \;=\; f(a^p_{\ell j}) \qquad \text{where} \qquad a^p_{\ell j} = \theta_{\ell j} + \sum_{i=1}^{N_{\ell-1}} w_{\ell ji}\, u^p_{\ell-1,i} \qquad (\ell = 2 \cdots L) \tag{C.1}$$

The neuron transfer function is usually the sigmoid function:

$$f(x) \;=\; \frac{1}{1 + e^{-x}} \tag{C.2}$$

A useful fact about this function is that $f'(x) = f(x)(1 - f(x))$. The MLP is trained using *backpropagation*. The object of backpropagation is to adjust the weights so that, for all training patterns, the sum squared difference between the desired and actual output (the error) is minimized, i.e.:

$$\text{minimize:} \qquad E \;=\; \sum_{p=1}^{P} E_p \tag{C.3}$$

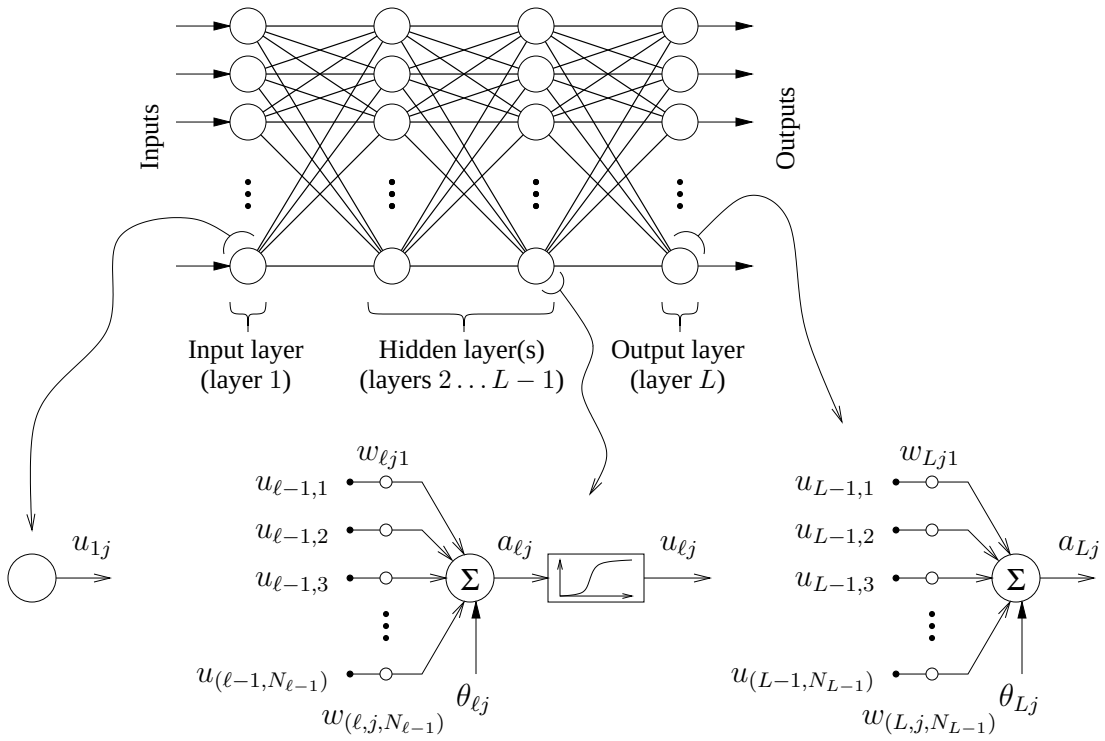


Figure C.1: *The multi-layer perceptron neural network.*

where

$$E_p = \sum_{i=1}^{N_L} \left(a_{Li}^p - d_i^p\right)^2 \tag{C.4}$$

The weights are adjusted by gradient descent:

$$w_{\ell ji} \leftarrow w_{\ell ji} - \alpha \frac{\partial E}{\partial w_{\ell ji}} \qquad (\alpha \text{ is the learning rate}) \tag{C.5}$$

$$\leftarrow w_{\ell ji} - \alpha \sum_{p=1}^{P} \frac{\partial E_p}{\partial w_{\ell ji}} \tag{C.6}$$

$$\leftarrow w_{\ell ji} - \alpha \begin{cases} \displaystyle\sum_{p=1}^{P} \frac{\partial E_p}{\partial u_{\ell j}^p} \cdot \frac{\partial u_{\ell j}^p}{\partial w_{\ell ji}} & (\text{if} \quad \ell = 2 \cdots (L-1)) \\ \displaystyle\sum_{p=1}^{P} \frac{\partial E_p}{\partial a_{Lj}^p} \cdot \frac{\partial a_{Lj}^p}{\partial w_{Lji}} & (\text{if} \quad \ell = L) \end{cases} \tag{C.7}$$

Now,

$$\frac{\partial u_{\ell j}^p}{\partial w_{\ell ji}} = \frac{\partial a_{\ell j}^p}{\partial w_{\ell ji}} f'(a_{\ell j}^p) \tag{C.8}$$

$$= u_{\ell-1,i}^p f'(a_{\ell j}^p) \tag{C.9}$$

$$= u_{\ell-1,i}^p u_{\ell j}^p (1 - u_{\ell j}^p) \tag{C.10}$$

Similarly,

$$\frac{\partial\, a^p_{Lj}}{\partial\, w_{Lji}} \;=\; u^p_{L-1,i} \tag{C.11}$$

Using the chain rule the $\partial E/\partial u$ of layer ℓ are associated with the $\partial E/\partial u$ of layer $\ell+1$:

$$D^p_{\ell j} = \frac{\partial\, E_p}{\partial\, u^p_{\ell j}} \;=\; \sum_{m=1}^{N_{\ell+1}} \frac{\partial\, E_p}{\partial\, u^p_{\ell+1,m}} \cdot \frac{\partial\, u^p_{\ell+1,m}}{\partial\, u^p_{\ell j}} \qquad \text{for } \ell = 1 \cdots (L-2) \tag{C.12}$$

$$= \sum_{m=1}^{N_{\ell+1}} \frac{\partial\, E_p}{\partial\, u^p_{\ell+1,m}} \cdot \frac{\partial\, a^p_{\ell+1,m}}{\partial\, u^p_{\ell j}} \cdot f'(a^p_{\ell+1,m}) \tag{C.13}$$

$$= \sum_{m=1}^{N_{\ell+1}} \frac{\partial\, E_p}{\partial\, u^p_{\ell+1,m}} \cdot w_{\ell+1,m,j} \cdot u^p_{\ell+1,m}(1 - u^p_{\ell+1,m}) \tag{C.14}$$

And similarly

$$\frac{\partial\, E_p}{\partial\, u^p_{L-1,j}} \;=\; \sum_{m=1}^{N_L} \frac{\partial\, E_p}{\partial\, a^p_{Lm}} \cdot w_{Lmj} \tag{C.15}$$

For the output layer the $\partial E/\partial a$ can be computed directly:

$$\frac{\partial\, E_p}{\partial\, a^p_{Lj}} \;=\; \frac{\partial \left[\sum_{i=1}^{N_L} \Big(a^p_{Li} - d^p_i\Big)^2\right]}{\partial\, a^p_{Lj}} \tag{C.16}$$

$$= 2\Big(a^p_{Lj} - d^p_j\Big) \tag{C.17}$$

The procedure for computing $\partial E/\partial\theta_{\ell j}$ is similar. The entire backpropagation algorithm is thus:

- Compute all $a^p_{\ell j}$ and $u^p_{\ell j}$ by forward propagation.
- For $p = 1 \cdots P$ and $j = 1 \cdots N_{L-1}$, compute

$$D^p_{L-1,j} = 2 \sum_{m=1}^{N_L} \Big(a^p_{Lm} - d^p_m\Big) w_{Lmj}$$

- For $p = 1 \cdots P$ and $\ell = (L-2) \cdots 2$ and $j = 1 \cdots N_\ell$, compute

$$D^p_{\ell j} = \sum_{m=1}^{N_{\ell+1}} D^p_{\ell+1,m} \cdot w_{\ell+1,m,j} \cdot u^p_{\ell+1,m}(1 - u^p_{\ell+1,m})$$

- For $j = 1 \cdots N_L$ and $i = 1 \cdots N_{L-1}$,

$$w_{Lji} \leftarrow w_{Lji} - 2\alpha \sum_{p=1}^{P} (a_{Lj} - d^p_j) u^p_{L-1,i}$$

- For $j = 1 \cdots N_L$,

$$\theta_{Lj} \leftarrow \theta_{Lj} - 2\alpha \sum_{p=1}^{P} \left(a_{Lj}^p - d_j^p \right)$$

- For $\ell = (L-1) \cdots 2$ and $j = 1 \cdots N_\ell$ and $i = 1 \cdots N_{\ell-1}$,

$$w_{\ell ji} \leftarrow w_{\ell ji} - \alpha \sum_{p=1}^{P} D_{\ell j}^p u_{\ell-1,i}^p u_{\ell j}^p (1 - u_{\ell j}^p)$$

- For $\ell = (L-1) \cdots 2$ and $j = 1 \cdots N_\ell$,

$$\theta_{\ell j} \leftarrow \theta_{\ell j} - \alpha \sum_{p=1}^{P} D_{\ell j}^p u_{\ell j}^p (1 - u_{\ell j}^p)$$

Backpropagation gets its name from the fact that the quantities $D_{\ell j}^p$ are computed by propagating their values backwards through the network.

Appendix D

Dynamic programming

D.1 The problem

Dynamic programming is an approach that allows systems of the form shown in figure D.1 to be optimized (for example this could be an unfolded dynamic control system). Dynamic programming selects the variables $x_1 \ldots x_n$ to minimize (or, with trivial modifications, maximize) the cost function

$$C = \sum_{i=1}^{n} c_i \tag{D.1}$$

where

$$y_{i+1} = T_i(x_i, y_i) \tag{D.2}$$
$$c_i = f_i(x_i, y_i) \tag{D.3}$$

Usually either y_1 or y_{n+1} is known beforehand. Note that the x_i and y_i can be scalars or vectors, and the c_i values are scalars.

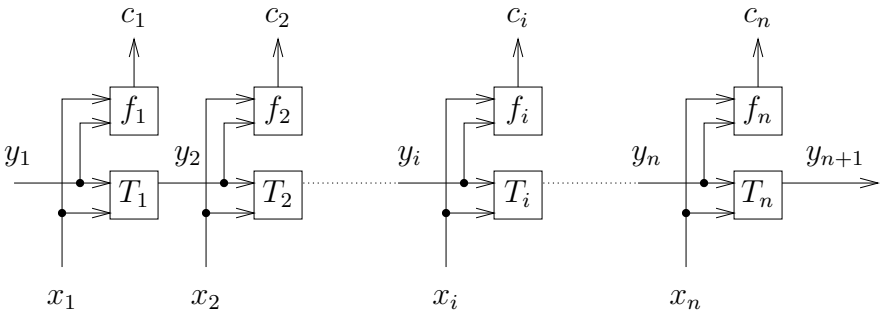


Figure D.1: *A system that can be optimized by dynamic programming.*

D.2 The solution

The system is decomposed using "Bellman's principle of optimality". This principle states that "an optimal solution is made up of optimal sub-solutions". This means that, if the variables $x_k \cdots x_n$ have been selected (with y_k fixed) to optimize the cost $\sum_{i=k}^{n} c_i$, then the variables $x_{k+1} \cdots x_n$ (with y_{k+1} keeping its same value) will be guaranteed to optimize the cost $\sum_{i=k+1}^{n} c_i$. This principle is implemented by defining the functions p and q:

- $p_i(y_i)$ is the value of x_i that is required for the given y_i to produce an optimal cost ($c_i + \cdots + c_n$), assuming that the variables $x_{i+1} \cdots x_n$ are chosen optimally.
- $q_i(y_i)$ is the cost $c_i + \cdots + c_n$ for the given y_i, assuming that $x_i \cdots x_n$ are chosen optimally.

Dynamic programming solves the problem backwards using recurrence relations. That is, given $p_{i+1}(y_{i+1})$ and $q_{i+1}(y_{i+1})$, $p_i(y_i)$ can be determined as the value of x_i that minimizes c_i plus the optimum subsequent cost $q_{i+1}(y_{i+1})$. The minimum cost[1] that is found is $q_i(y_i)$. Thus:

$$p_i(y_i) = \min_{(x_i)} \left[f_i(x_i, y_i) + q_{i+1}(T_i(x_i, y_i)) \right] \tag{D.4}$$

$$q_i(y_i) = f_i(p_i(y_i), y_i) + q_{i+1}(T_i(p_i(y_i), y_i)) \tag{D.5}$$

To use these recurrence relations, first define $q_{n+1}(y_{n+1})$. A backwards pass is made for $i = n \cdots 1$: first $p_i(y_i)$ is computed and then $q_i(y_i)$ is computed. Then a forward pass is made for $i = 1 \cdots n$: first $x_i = p_i(y_i)$ is computed and then $y_{i+1} = T(x_i, y_i)$ is computed. When the functions p and q are computed, what is actually calculated is a set of parameters that determine the functions. What these parameters actually represent will depend on the problem being solved. The $p(\cdot)$ parameters are needed on the forward pass, so they must be saved by the backwards pass. The $q(\cdot)$ parameters are not needed on the forward pass, so their storage can be re-used on the backwards pass.

D.3 A linear controller example

The above principles will now be applied to find the optimal control inputs for a simple discrete-time linear control system:

$$\mathbf{y}_{i+1} = A\mathbf{y}_i + Bx_i \tag{D.6}$$

$$c_i = (C\mathbf{y}_i - d_i)^2 + Dx_i^2 \tag{D.7}$$

where $\mathbf{y}_i$ is the $n_y \times 1$ system state vector, x_i is the (scalar) control force input and d_i is the desired (scalar) value for an element of $\mathbf{y}_i$ at each time step. The control goal specified by the cost function is to get an element of $\mathbf{y}_i$ to follow the reference d_i without making the control effort x_i too large. A and B are the system matrices, C is a $1 \times n_y$ vector that selects an element of $\mathbf{y}$, and D is a scalar constant that quantifies the relative importance of minimizing $|x_i|$. Define

$$q_{n+1} = (C\mathbf{y}_{n+1} - d_{n+1})^2 \tag{D.8}$$

and assume that the function q has the form

$$q_i(\mathbf{y}_i) = \mathbf{y}_i^T E_i \mathbf{y}_i + F_i \mathbf{y}_i \tag{D.9}$$

[1] Note that $\min_{(x)} f(x)$ is defined to be the value of x that minimizes $f(x)$.

where E_i is a $n_y \times n_y$ matrix and F_i is a $1 \times n_y$ vector. Thus:

$$\begin{aligned} p_i(\mathbf{y}_i) &= \min_{(x_i)} \left[f_i(x_i, \mathbf{y}_i) + q_{i+1}(T_i(x_i, \mathbf{y}_i)) \right] && \text{(D.10)} \\ &= \min_{(x_i)} \Big[(C\mathbf{y}_i - d_i)^2 + Dx_i^2 + && \text{(D.11)} \\ &\qquad (A\mathbf{y}_i + Bx_i)^T E_{i+1}(A\mathbf{y}_i + Bx_i) + F_{i+1}(A\mathbf{y}_i + Bx_i) \Big] \\ &= Q_{i+1}\mathbf{y}_i + R_{i+1} && \text{(D.12)} \end{aligned}$$

where

$$\begin{aligned} Q_i &= \frac{-B^T E_i A}{D + B^T E_i B} && \text{(D.13)} \\ R_i &= -\frac{F_i B}{2(D + B^T E_i B)} && \text{(D.14)} \end{aligned}$$

Now,

$$\begin{aligned} q_i(\mathbf{y}_i) &= f_i(p_i(\mathbf{y}_i), \mathbf{y}_i) + q_{i+1}(T_i(p_i(\mathbf{y}_i), \mathbf{y}_i)) && \text{(D.15)} \\ &= (C\mathbf{y}_i - d_i)^2 + D(Q_{i+1}\mathbf{y}_i + R_{i+1})^2 + && \text{(D.16)} \\ &\quad (A\mathbf{y}_i + B(Q_{i+1}\mathbf{y}_i + R_{i+1}))^T E_{i+1}(A\mathbf{y}_i + B(Q_{i+1}\mathbf{y}_i + R_{i+1})) + \\ &\quad F_{i+1}(A\mathbf{y}_i + B(Q_{i+1}\mathbf{y}_i + R_{i+1})) \\ &= \mathbf{y}_i^T E_i \mathbf{y}_i + F_i \mathbf{y}_i && \text{(D.17)} \end{aligned}$$

where

$$\begin{aligned} E_i &= C^T C + DQ_{i+1}^T Q_{i+1} + ((A + BQ_{i+1})^T E_{i+1}(A + BQ_{i+1})) && \text{(D.18)} \\ F_i &= -2d_i C + 2DR_{i+1}Q_{i+1} + ((A + BQ_{i+1})^T E_{i+1}(BR_{i+1}))^T + && \text{(D.19)} \\ &\quad ((BR_{i+1})^T E_{i+1}(A + BQ_{i+1})) + F_{i+1}A + F_{i+1}BQ_{i+1} \end{aligned}$$

Thus the algorithm for computing the optimal control strategy is:

- Set $E_{n+1} = C^T C$ and $F_{n+1} = 2d_{n+1}C$.
- For $i = n \cdots 1$:
  - Set Q_{i+1} and R_{i+1} according to equation D.13 and equation D.14.
  - Set E_i and F_i according to equation D.18 and equation D.19.
- For $i = 1 \cdots n$:
  - Set $x_i = Q_{i+1}\mathbf{y}_i + R_{i+1}$
  - Compute y_{i+1} from equation D.6.

Despite the simplicity and power of the dynamic programming approach, it is not practical to directly implement it in a real controller. It is not an on-line algorithm because it requires a backwards pass before the control inputs can be computed. Also, it requires exact knowledge of the system dynamics (the matrices A and B).

Appendix E

Eligibility profile cookbook

The eligibility profile function is

$$\epsilon_i \;=\; C\,A^{i-1}\,B \qquad (i \geq 1) \tag{E.1}$$

A useful class of second order eligibility profiles is parameterized by the numbers k_a and k_b, and has the following matrices:

$$A \;=\; \begin{bmatrix} 1 & h \\ -hk_a & 1-hk_b \end{bmatrix} \tag{E.2}$$

$$B \;=\; \begin{bmatrix} 0 \\ hk_a \end{bmatrix} \tag{E.3}$$

$$C \;=\; \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{E.4}$$

The matrices are derived from an Euler approximation (with step size h) of the following continuous differential equation

$$\ddot{\varepsilon} \;=\; k_a\,(s-\varepsilon) - k_b\,\dot{\varepsilon} \tag{E.5}$$

$$\varepsilon(0) \;=\; 0 \tag{E.6}$$

$$\dot{\varepsilon}(0) \;=\; h\,k_a \tag{E.7}$$

$$\text{where} \qquad \epsilon_i \;=\; \varepsilon(ih) \tag{E.8}$$

Note that $\dot{\varepsilon}(0) = hk_a$ because the driving signal $s(t)$ is equal to the impulse $h\delta(t)$ (where $\delta(t)$ is the Dirac-delta function). The solutions for $\varepsilon(t)$ are

$$q \;=\; k_b^2 - 4\,k_a \tag{E.9}$$

$$\varepsilon(t) \;=\; \begin{cases} \dfrac{k_a}{\sqrt{q}} e^{(-k_b+\sqrt{q})t/2} - \dfrac{k_a}{\sqrt{q}}\, e^{(-k_b-\sqrt{q})t/2} & (\text{if} \quad q>0) \\[2ex] \dfrac{2k_a}{\sqrt{-q}} e^{-k_b t/2} \sin\left(\dfrac{t\sqrt{-q}}{2}\right) & (\text{if} \quad q<0) \\[2ex] k_a t\, e^{-k_b t/2} & (\text{if} \quad q=0) \end{cases} \tag{E.10}$$

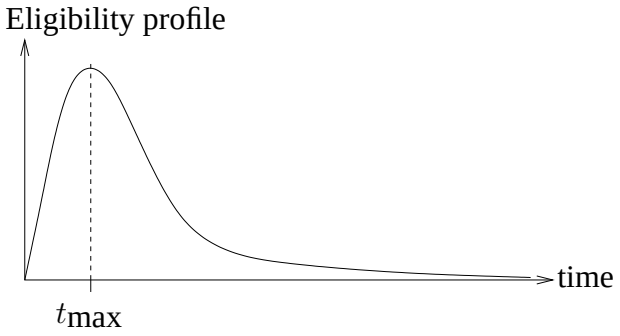


Figure E.1: *A second order critically damped eligibility profile.*

These solutions are respectively over-damped ($q > 0$), under-damped ($q < 0$) and critically damped ($q = 0$). The following table shows how to select k_a and k_b for these three cases.

Mode	Prototype profile	Parameters
Over-damped	$e^{-\tau_1 t} - e^{-\tau_2 t}$	$k_a = \tau_1\tau_2$ and $k_b = \tau_1 + \tau_2$
Under-damped	$e^{-\tau t}\sin(\omega t)$	$k_a = \tau^2 + \omega^2$ and $k_b = 2\tau$
Critically damped	$t\, e^{-\tau t}$	$k_a = \tau^2$ and $k_b = 2\tau$

For the critically damped case another useful way to set the parameters is by $t_{\max}$, the time at which the eligibility profile reaches a maximum (see figure E.1). In this case

$$k_a = \frac{1}{t_{\max}^2} \tag{E.11}$$

$$k_b = \frac{2}{t_{\max}} \tag{E.12}$$

Appendix F

Trace precision

F.1 The problem

The trace value λ_j is computed by the FOX algorithm:

$$\lambda_j = \sum_{i=1}^{j} e_i \, C \, A^i \tag{F.1}$$

This value is used to compute the quantity

$$\Delta = (\lambda_{i_2} - \lambda_{i_1 - 1}) A^{-i_1} \qquad (i_2 > i_1) \tag{F.2}$$

The exponentially decreasing A^i values causes λ_j to quickly converge to a steady value for large j. As time (i_1 and i_2) increases, equation F.2 multiples an exponentially increasing value (A^{-i_1}) by an exponentially decreasing difference ($\lambda_{i_2} - \lambda_{i_1 - 1}$). Thus Δ quickly loses precision and can cause numerical errors in the FOX algorithm.

This appendix will quantify the amount of precision lost. The worst case precision loss is calculated, so it will be assumed that $e_i = 1$, and that $i_2 = \infty$ and $i_1 = j + 1$ (with $j \gg 1$).

F.2 The scalar case

First the precision lost when computing $(\lambda_\infty - \lambda_j)\vartheta^{-(j+1)}$ will be calculated, where

$$\lambda_j = \sum_{i=1}^{j} \vartheta^i \qquad , |\vartheta| < 1 \tag{F.3}$$

$$= \frac{\vartheta(\vartheta^j - 1)}{\vartheta - 1} \tag{F.4}$$

$$\text{and} \qquad \lambda_\infty = \frac{\vartheta}{1 - \vartheta} \tag{F.5}$$

Now, λ_∞ and λ_j have a similar magnitude because $j \gg 1$, so the decimal precision lost in the difference $\lambda_\infty - \lambda_j$ is (referring to figure F.1):

$$\text{precision lost} = k_1 - k_2 \tag{F.6}$$

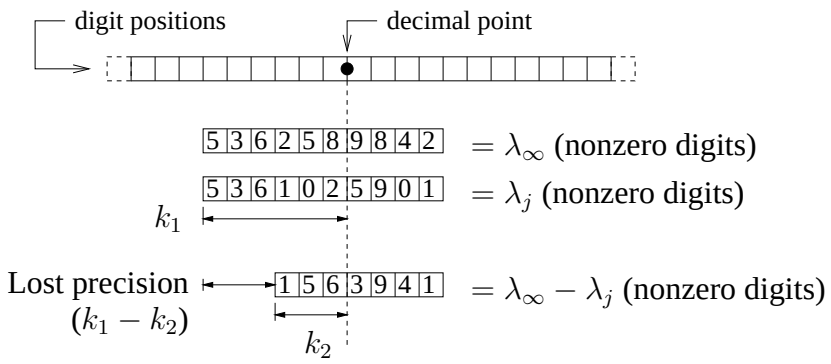


Figure F.1: *Calculating the precision lost in the difference $\lambda_\infty - \lambda_j$.*

$$\begin{aligned}
&\approx \log_{10}(\lambda_\infty) - \log_{10}(\lambda_\infty - \lambda_j) && \text{(F.7)}\\
&= \log_{10}\left(\frac{\lambda_\infty}{\lambda_\infty - \lambda_j}\right) && \text{(F.8)}\\
&= \log_{10}\left(\frac{1}{\vartheta^{j+1}}\right) && \text{(F.9)}\\
&= -(j+1)\log_{10}\vartheta && \text{(F.10)}\\
&\approx j(-\log_{10}\vartheta) && \text{(F.11)}
\end{aligned}$$

When all of the difference's precision has been lost, numerical error will dominate the result.

F.3 The matrix case

The matrix A can be decomposed into a diagonal matrix Λ of eigenvalues and a matrix S of eigenvectors as follows:

$$A = S\,\Lambda\,S^{-1} \qquad \text{(F.12)}$$

Thus

$$\begin{aligned}
\lambda_j &= \sum_{i=1}^{j} C\,A^i && \text{(F.13)}\\
&= \sum_{i=1}^{j} C\,(S\,\Lambda^i\,S^{-1}) && \text{(F.14)}\\
&= C\,S\left[\sum_{i=1}^{j}\Lambda^i\right]S^{-1} && \text{(F.15)}\\
&= C\,S\begin{bmatrix} \sum_{i=1}^{j}\vartheta_1^i & & \cdots & 0 \\ & \sum_{i=1}^{j}\vartheta_2^i & & \\ \vdots & & \ddots & \\ 0 & & & \sum_{i=1}^{j}\vartheta_n^i \end{bmatrix}S^{-1} && \text{(F.16)}
\end{aligned}$$

where $\vartheta_1 \ldots \vartheta_n$ are the eigenvalues of A (they should all have magnitude $|\vartheta| < 1$ for the FOX algorithm). Each diagonal in the matrix is equivalent to an instance of equation F.3. From this it is apparent that the first diagonal element to lose precision will cause the entire value of Δ to lose precision. From equation F.11 this first element will be the one with the smallest (i.e. least magnitude) eigenvalue. It is curious that this smallest eigenvalue has the shortest lived effect on the eligibility profile, but the largest effect on numerical accuracy.

F.4 Trace buffer size

Let ϑ_s be the smallest (in magnitude) eigenvalue of A. The trace buffer size σ is selected from a parameter p such that, at most, no more than p decimal digits of precision are lost. From equation F.11:

$$p = \sigma\left(-\log_{10}\vartheta_s\right) \tag{F.17}$$

so

$$\text{maximum } \sigma = -\frac{p}{\log_{10}\vartheta_s} \tag{F.18}$$

$$= \frac{-p\ \ln 10}{\ln\vartheta_s} \tag{F.19}$$

The maximum eligibility decay (relative to its initial value) at the end of the trace buffer is (assuming that all the eigenvalues are "close" to ϑ_s) :

$$\text{maximum decay} = \vartheta_s^{\sigma} \tag{F.20}$$

$$= \left(10^{-p/\sigma}\right)^{\sigma} \tag{F.21}$$

$$= \frac{1}{10^p} \tag{F.22}$$

For example a good compromise is to select $p = 2$ (two digits of precision lost and a maximum decay of 0.01). Then:

$$\text{maximum } \sigma = \frac{-4.605}{\ln\vartheta_s} \tag{F.23}$$

For reference, the IEEE single precision floating point data type has 7.2 decimal digits of precision (24 bits of mantissa) so the loss of two digits of precision is no problem.

Appendix G

Eligibility order reduction

The eligibility profile function is

$$\epsilon_i \;=\; C\,A^{i-1}\,B \qquad (i \geq 1) \tag{G.1}$$

where A has size $n_y \times n_y$, B has size $n_y \times n_x$, C has size $1 \times n_y$ and ϵ_i is a scalar.

It is sometimes necessary to reduce the order of this model by eliminating small eigenvalues. Small eigenvalues play an insignificant role in determining ϵ_i but they can cause great loss of numerical precision in the FOX algorithm. New, smaller matrices $\overline{A}$, $\overline{B}$ and $\overline{C}$ must be found such that $\overline{\epsilon}_i$ approximates ϵ_i:

$$\begin{aligned} \overline{\epsilon}_i &= \overline{C}\,\overline{A}^{\,i-1}\,\overline{B} \\ &\approx \epsilon_i \end{aligned} \tag{G.2}$$

The matrix A can be decomposed into a diagonal matrix Λ of eigenvalues and a matrix S of eigenvectors as follows:

$$A \;=\; S\,\Lambda\,S^{-1} \tag{G.3}$$

which gives

$$\epsilon_i \;=\; [C\,S]\,\Lambda^{i-1}\,[S^{-1}\,B] \tag{G.4}$$

For example, if

$$A \;=\; \begin{bmatrix} 1 & 0 & 0.1 & 0 \\ 0 & 1 & 0 & 0.1 \\ -0.08 & 0.06 & 0.7 & 0.2 \\ 0.1 & -0.1 & 0 & 1 \end{bmatrix} \tag{G.5}$$

(G.6)

then

$$\Lambda \;=\; \begin{bmatrix} 0.9847{+}0.0520i & 0 & 0 & 0 \\ 0 & 0.9847{-}0.0520i & 0 & 0 \\ 0 & 0 & 0.9718 & 0 \\ 0 & 0 & 0 & 0.7589 \end{bmatrix} \tag{G.7}$$

$$S = \begin{bmatrix} 0.4838{-}0.2314i & 0.4838{+}0.2314i & -0.7061 & -0.3790 \\ 0.6770{-}0.1641i & 0.6770{+}0.1641i & -0.6539 & -0.0556 \\ 0.0462{+}0.2870i & 0.0462{-}0.2870i & 0.1993 & 0.9139 \\ -0.0184{+}0.3772i & -0.0184{-}0.3772i & 0.1846 & 0.1341 \end{bmatrix} \tag{G.8}$$

The rows and columns of Λ and S corresponding to the smallest eigenvalues are removed, to obtain $\overline{\Lambda}$ and $\overline{S}$. In this example the fourth row and column are removed, corresponding to the smallest eigenvalue 0.7589 :

$$\overline{\Lambda} = \begin{bmatrix} 0.9847{+}0.0520i & 0 & 0 \\ 0 & 0.9847{-}0.0520i & 0 \\ 0 & 0 & 0.9718 \end{bmatrix} \tag{G.9}$$

$$\overline{S} = \begin{bmatrix} 0.4838{-}0.2314i & 0.4838{+}0.2314i & -0.7061 \\ 0.6770{-}0.1641i & 0.6770{+}0.1641i & -0.6539 \\ 0.0462{+}0.2870i & 0.0462{-}0.2870i & 0.1993 \end{bmatrix} \tag{G.10}$$

Now, for the remaining eigenvalues and eigenvectors to correctly combine to give $\overline{\epsilon}_i$, the reduced order model must be

$$\overline{\epsilon}_i = \overline{[C\ S]}\,\Lambda^{i-1}\,\overline{[S^{-1}\ B]} \tag{G.11}$$

where $\overline{[C\ S]}$ is the matrix CS with the chosen rows and columns removed, and similarly for $\overline{[S^{-1}\ B]}$. From equation G.4 the following must also be true:

$$\overline{\epsilon}_i = [\overline{C}\ \overline{S}]\,\overline{\Lambda}^{\,i-1}\,[\overline{S}^{-1}\ \overline{B}] \tag{G.12}$$

So by comparing equation G.12 and equation G.11 the reduced order model is

$$\overline{B} = \overline{S}\,\overline{[S^{-1}\ B]} \tag{G.13}$$

$$\overline{C} = \overline{[C\ S]}\,\overline{S}^{-1} \tag{G.14}$$

and

$$\overline{A} = \overline{S}\,\overline{\Lambda}\,\overline{S}^{-1} \tag{G.15}$$

Appendix H

Software systems

H.1 Introduction

Several software systems were developed by the author to support the work described in this thesis. They are written in C++ and Perl, and run under the Linux/X-Windows environment. The software modules and files used to support intelligent robot control simulation are shown in figure H.1. Briefly, they are:

- `biped.dnd` describes the dynamic system that is the robot's "brain", in DND (dynamic network description) language (see Appendix J).
- `dnd2exe`, a program which compiles DND into C source code, producing the file `biped.c` (it is part of the `Dyson` system described in Appendix J).
- The system compiler `gcc` compiles `biped.c` into `biped.so`, a shared library object.
- `biped.cfg` describes the mechanical structure of the robot, and also contains some simulation parameters. This file is used by the RoboDyn library.
- `CyberSim` perform the actual simulation.
- `cybersim-results.rsdf` is a binary data file that contains the state (position and velocity of all the mechanical elements) at each time step.
- `dsdata.rsdf` is a binary data file that contains the internal state of the "brain" dynamic system at each time step.
- `CyberView` is an interactive rendering program which allows the user to move through a virtual 3D environment and observe the operation of the robot over time.
- `DSVIEW` is an interactive application which allows the user to graphically view the dynamic system variables over time.
- `RoboDyn` is a library which performs the dynamical mechanical simulation of the robot.
- `FoxN` is a library which implements the FOX algorithm.
- `R3D` is a library which performs 3D rendering.

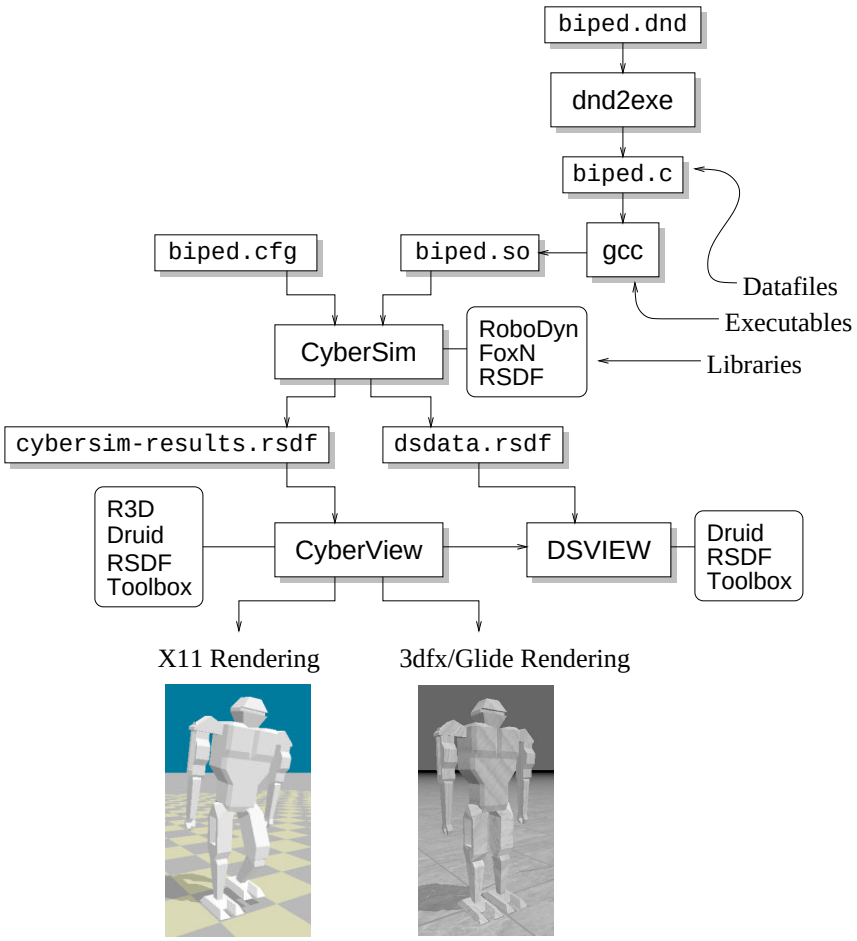


Figure H.1: *The software modules and files used to support robot intelligent control simulation.*

- `Druid` is a GUI class library and widget set.
- `RSDF` is a library which reads and writes files in the RSDF format.
- `Toolbox` is a library which implements garbage collection, runtime type identification (RTTI), exception handling, interfaces, and hash tables.

The major systems are briefly described in this appendix, except for `dnd2exe` (which is described in Appendix J) and `RoboDyn` (which is described in Appendix I)).

H.2 CyberSim

CyberSim performs 3D mechanical simulation of a robot together with its intelligent controller. It brings together the `Dyson`, `FoxN`, `RoboDyn` and `RSDF` libraries. It takes two files as input: a `RoboDyn` configuration file which describes the mechanical structure of the robot, and an object file which is dynamically linked to CyberSim to implement control functions. The configuration file also specifies the simulation time step size, the number of iterations to perform and the integration method (Euler, midpoint or Runge-Kutta fourth order). It generates data files in RSDF format containing the mechanical and controller system states at every time step.

H.3 CyberView and DSVIEW

CyberView (figure H.2) is an interactive rendering program which allows the user to move through a virtual 3D environment and observe the operation of mechanisms simulated with CyberSim. It displays the objects stored in CyberSim data files. The custom graphics engine (the `R3D` library) supports flat shaded rendering to an X-terminal or texture mapped real-time rendering to a 3dfx/Voodoo graphics accelerator card.

CyberView allows an external data display (EDD) program to be invoked and synchronized with CyberView's currently displayed time step. One such program is DSVIEW (figure H.3) which displays the internal variables of the dynamic system that controls the mechanical model (see Appendix J).

CyberView's main window is shown in figure H.2. The user can move around the virtual environment by clicking and dragging in the 3D display area (the mouse button that is pressed determines the set of axes that are moved along). The simulation movie can be played using video recorder style buttons. The menu options allow the user to open and close data files, start an EDD program, save a single frame or a sequence of frames as PPM graphics files, change the lighting angle, set and jump to up to 10 camera positions, render in wire-frame or solid mode, and set the shadow and z-buffer parameters.

H.4 R3D

The `R3D` library used by CyberView performs the following tasks: representation of the drawing primitives (3D polyhedral objects and infinite planes), geometric transformation between object space and screen space, clipping of drawing primitives to the viewing frustrum, lighting calculations, z-buffering for X11 rendering, texture mapping computations and shadow mapping.

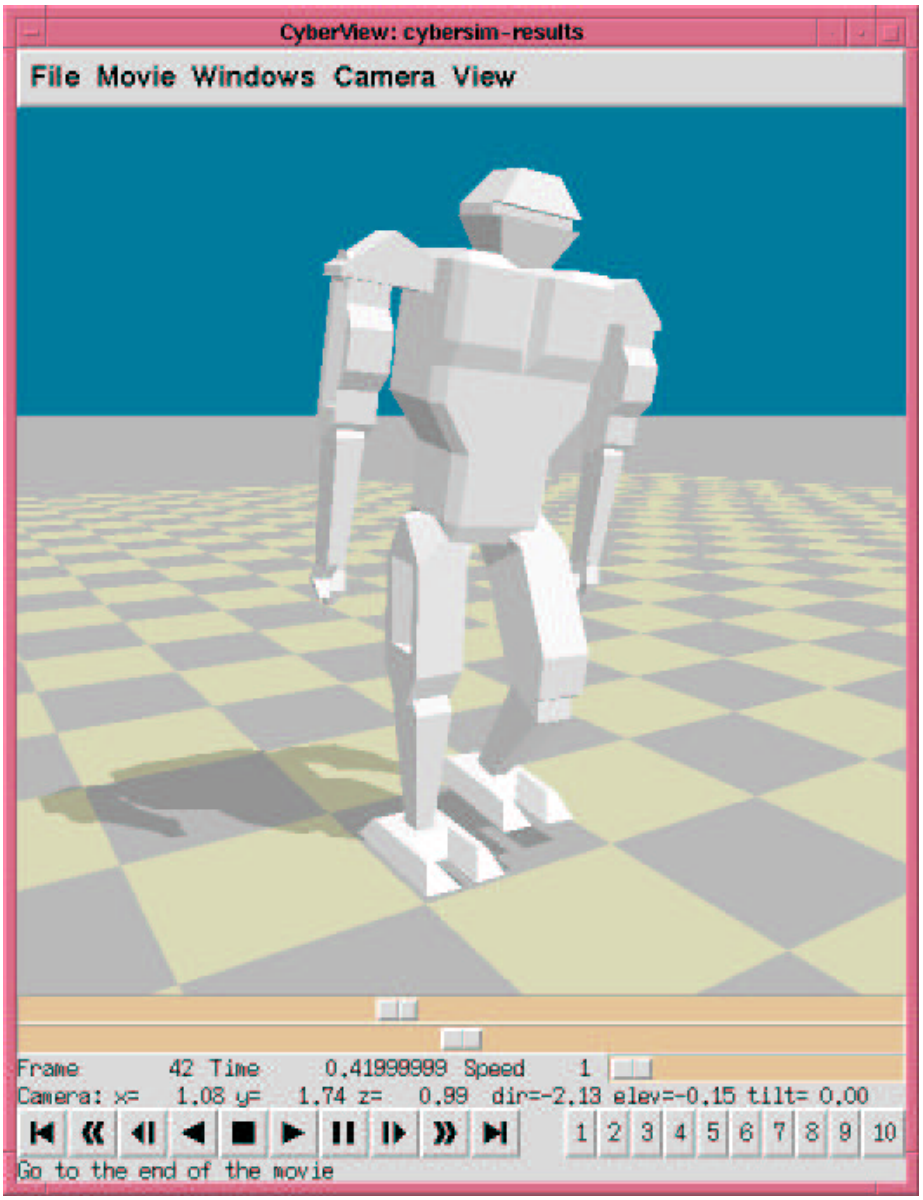


Figure H.2: *The CyberView main window.*

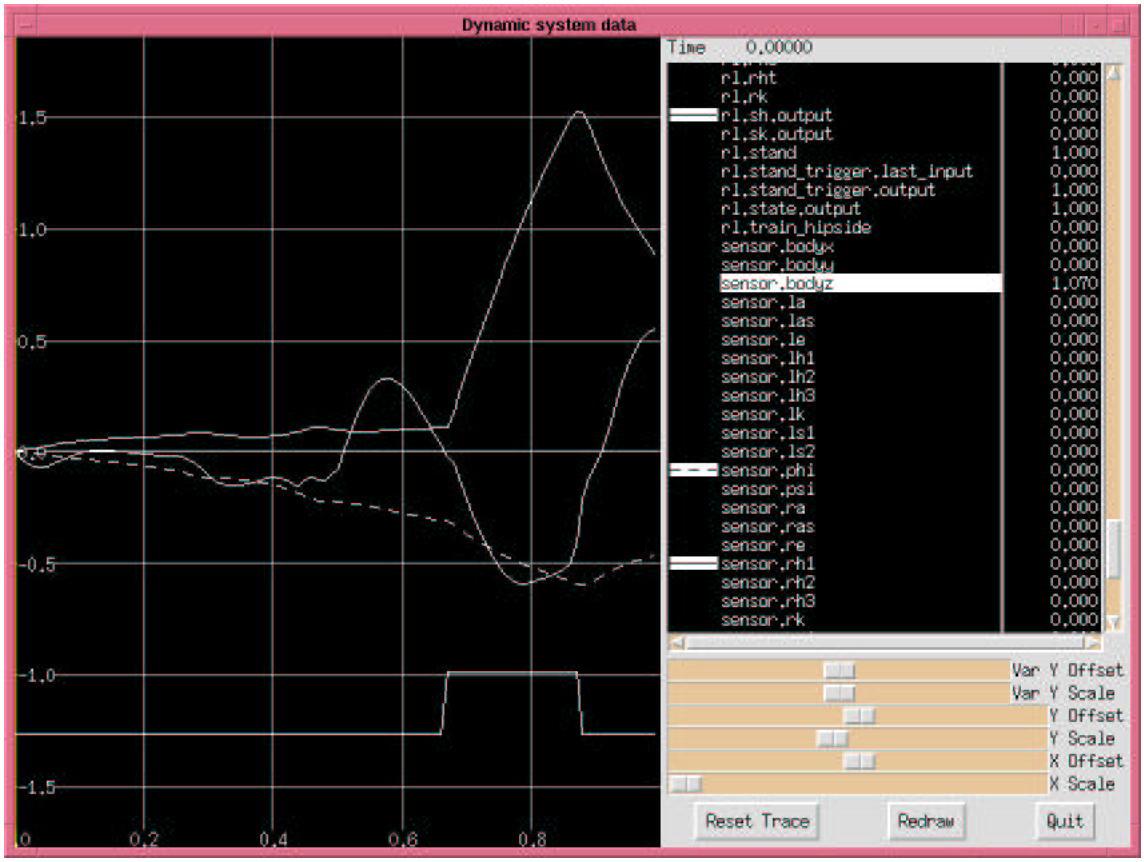


Figure H.3: *DSVIEW: The dynamic system viewer main window.*

H.4.1 The multi-resolution Z-buffer algorithm

When `R3D` renders to an X11 window, it draws flat-shaded polygons with hidden surface removal. A very simple hidden surface algorithm is to give each polygon a depth map and use the z-buffer algorithm [35].

However, with the normal z-buffer algorithm every pixel in every polygon must have its depth tested against the z-buffer. In software this is very slow compared to the speed at which an accelerated graphics card can render pixels. Note that anything more complicated than just flat-shaded polygons needs to be rendered (like Gouraud shading or texture mapping) then the z-buffer speed problem is insignificant compared to the extra processing required.

The `R3D` library uses a "multi-resolution z-buffer" algorithm so that it only has to do the full resolution (or fine resolution) z-buffer process on small blocks of the image, which mainly correspond to places where more than two polygons intersect. Large areas of constant color are drawn quickly without z-buffer computations.

Say the image size is 512×512 pixels. First the polygons are rendered on to a coarse grid of, say, 32×32 cells. Cells are classified as either:

1. Unused.
2. Completely filled by one polygon.
3. Filled by more than one polygon.

Type-1 cells are ignored. Type-2 cells are immediately filled with the appropriate color. Type-3 cells get the full z-buffer treatment. The majority of pixels in the image are hopefully going to be in type-1 or type-2 cells, so they can be processed quickly. Type-3 cells (where full z-buffering occurs) are only needed in the interior edges of an image. As the image complexity increases (i.e. more edges) then the effectiveness of this approach will be decreased. It will have the greatest improvement over z-buffer in cases where there are relatively few, large polygons in the image. An image that has a large number of small polygons (e.g. a gridded object) may not be rendered very efficiently compared to straight z-buffer.

H.5 FoxN

The `FoxN` library implements the FOX algorithm described in Chapter 5. It defines several C++ classes:

- `TraceN`, which performs the eligibility trace computations.
- `FoxN`, which performs the CMAC mapping and the main part of the FOX algorithm.
- `GetAtoi`, which allows fast table-based matrix exponentiation (it is used to compute the A^i values).
- `HPFloat`, a high precision floating point number class, for performing experiments on FOX systems that suffer from the small-eigenvalue precision problem (see Appendix F).

H.6 RSDF

RSDF is an acronym for Realm Structured Data Format. It is a file format used to store CyberSim simulation results. The file format is specially constructed to get the following features:

- Flexibility: the format of the stored data is a nested series of structures and arrays. The structure/array definitions are specified in the file header.
- Reading speed: all data is properly aligned so it is easy to `mmap()` the file into memory and access it directly.
- Accessing ease: When `mmap()`ed, nested arrays can be accessed with a simple indexing calculation, e.g. the address of `x[i].y[j].z[k]` is `k1*i+k2*j+k3*k+k4`.

H.7 Druid

Druid is a C++ GUI widget library for the X-windows system. Druid is fast and small, and supports rapid application development with a complete set of GUI widgets. It was written to explore OOP design strategies, and is used extensively in the other systems. These are the classes provided by Druid:

`CComponent`	For reference counting, RTTI, interfaces, exceptions.
`CTextArray`	Container for text arrays.
`ZColor`	A read-only sharable color in the default colormap.
`ZDrawable`	Represents anythings that can be drawn into.
`ZPixmap`	An X11 Pixmap.
`ZWindow`	An X11 window.
`ZFont`	A font and associated metric information.
`Zgc`	Represents an X11 graphics context.
`ZPalette`	Standard palette.
`ZPlugin`	Abstract module to give a window extra behaviors.
`ZAutoHelp`	Automatic status-bar help.
`ZFocus`	Participate in keyboard focusing.
`ZPacker`	Geometry-packs sub-windows.
`ZPackInfo`	Information packet for `ZPacker`.
`ZScroller`	Scrollable sub-window.
`ZTopFocuser`	Top-level focus window.
`ZWidget`	A widget is a window with palette and font.
`ZButton`	A pushbutton...
`ZPixmapButton`	...that displays an image.
`ZTextButton`	...that displays text.
`ZFrame`	A 3D frame.
`ZLabel`	A static text label.
`ZLineEdit`	An editable text box.
`ZMenuBar`	A menu bar.
`ZScrollbar`	A scrollbar.
`ZSlider`	A position-slider.
`ZStatusLine`	A status line.
`ZTextList`	A list of text items.
`ZWidgetFont`	A plug-in widget font.
`ZPrivateColor`	A private read/write color cell.
`(XSetWindowAttributes)`	
`ZSetWindowAttributes`	Allows window attributes to be set.

`ZMenuBarItem`	A menu bar item.
`ZMenuItem`	A popup menu item.

Appendix I

RoboDyn

I.1 Introduction

RoboDyn is a C++ class library that simulates the dynamics (i.e. the motion) of articulated rigid bodies. The numerical engine is based on the PhD work of Scott McMillan [76, 77, 78, 79], which he implemented in the DynaMechs library. RoboDyn provides very efficient simulation: it is based on Featherstone's recursive articulated-body algorithm [34] which executes in $O(n)$ time where n is the number of links in the system. Numerical integration is user selected from Euler, midpoint or Runge-Kutta (in the future Runge-Kutta-Nyström methods should be implemented for extra speed [25]). This appendix gives some superficial details about the RoboDyn library.

I.2 Links, joints and the AB tree

An articulated rigid-body (AB) is a collection of "links" that are connected together by "joints". Each link is a solid body with its own mass and rotational inertia. Each joint is a constraint between two links so that they can only move in a certain way relative to each other.

Featherstone's AB algorithm requires that all the links are in a tree structure, with one link designated the system root. Closed loops of links are not allowed. This means that all links except the root have one inboard (or parent) link. So each link except the root has an associated joint which attaches it to its parent link. Each link has a coordinate system (CS) which is fixed relative to that link. The CS is transformed (rotated and translated) with respect to the CS of the parent link.

Each joint has between zero and six degrees of freedom. That is, the joint outboard link position is a function of n parameters of the inboard link position. Mobile root links have 6 joint parameters because they can be positioned arbitrarily in 3D space. Some special "spacer" links have no parameters because the joints are always in a fixed position.

The joint parameters are called the positional state variables. They parameterize the transformation from the inboard to the outboard CS. The time derivatives of the positional state variables are the velocity state variables. The simulation has n position and n velocity state variables, where n is the sum of the number of joint parameters for all links. The variables are arranged into an n-sized vector in a standard order which comes from a depth-first traversal of the AB tree.

Different link types in RoboDyn correspond to different types of transformations that get the link CS from the parent CS.

The articulated-body (AB) algorithm has three stages, which each recurse over the entire AB tree:

1. Forward kinematics: kinematic (positional) quantities are computed based on joint positions, from inboard links to outboard links.
2. Backward dynamics: some dynamic quantities such as the inboard reflected inertia and the inboard force are computed, from outboard links to inboard links.
3. Forward accelerations: the joint accelerations are computed, from inboard links to outboard links.

I.3 Widgets

The RoboDyn kernel computes the forces on the links that arise from the constraints at the joints. (In fact these forces are not computed explicitly, but the accelerations arising from them are). However, other "external" forces must be applied to the links and joints to make the simulation interesting. These include:

- Link-to-link and link-to-ground collision forces.
- Spring forces between links.
- Damping forces on the links, e.g. if the link is moving in a viscous medium.
- "Thruster" forces on the links, e.g. if a link is motivated by a turbine or reaction engine (rocket motor).
- Joint frictional forces.
- Joint limits (the stopping force when a joint comes to the end of its motion).
- Motor forces, using simple or detailed motor models.

Gravity is not on this list because it is handled specially in the root links.

In RoboDyn, "widgets" are used to apply forces to links or joints. Widgets can interact with the external force and joint force variables of the links. The forces applied must be a function of the system state (position and velocity) only, no accommodation is made for internal widget states. This restriction means it is always possible to compute the widget's potential (position-related) and dissipated (velocity-related) energies. The widgets are not allowed to have internal state because that would greatly complicate the structure of the library, i.e. allowances would have to be made for arbitrary dynamical system structures.

Each widget can have a number of actuator and sensor variables. These variables are supplied/read by the user of RoboDyn at each time step. The actuators parameterize the widget effects: for example, actuators could be thruster force or motor signals. The sensor values allow the user to get feedback from the widgets. For example, a sensor could be contact force for a collision detection widget.

I.4 RoboDyn configuration file

The general structure of the configuration file is very simple. There are only two kinds of data that can be written: numbers and strings. Strings are always quoted `"like this"`. There are two kinds of data structures: arrays and associations. Arrays are written by enclosing the values in parentheses. For example, here is a four element array:

```
( 1 "foo" 4.8e-2 "bar" )
```

Associations are written by enclosing key-value pairs in curly brackets, like this:

```
{ mass 1 size 4.0 name "foo" }
```

This associates `mass` with 1, `size` with 4.0, etc. The key values are unquoted names. In both arrays and associations the values can be other arrays and associations, so complex hierarchical data structures can be constructed.

The RoboDyn configuration file is just a single un-bracketed association. All links are stored as associations in an association called `Links`. All link widgets are stored as associations in an array called `LinkWidgets`. So the configuration file has this overall structure:

```
Links{
        link_1_name { type "Link1Type" outboard_links (...) ... }
        link_2_name { type "Link2Type" outboard_links (...) ... }
        ...
}
LinkWidgets (
        { type "Widget1Type" ... }      # first link widget
        { type "Widget2Type" ... }      # second link widget
        ...
)
```

Most RoboDyn classes initialize themselves from associations in the configuration file.

I.5 RoboDyn class hierarchy

`ABSystem`	An articulated body system. This is the container for the entire AB system, in a typical simulation this is the only object you will need to make.
`ArticulatedBody`	This represents a node in an AB tree. It contains pointers to the outboard (child) ABs. It represents the joint connecting this to the parent (or inboard) AB (or none if this is the system root).
`RigidBody`	This adds mass properties to the AB: the total mass, inertia tensor and center of mass position.
`MobileRootLink`	Mobile system root link.
`Link`	This class adds a few AB implementation variables to `RigidBody`, and has abstract definitions for all the coordinate system transformation functions.
`MDHLink`	An MDH link has a single degree of freedom joint. Its transformation is defined by modified Denavit-Hartenberg parameters.

`RevoluteLink`	Revolute, or hinge-type joint.
`PrismaticLink`	Prismatic, or piston-type joint.
`BallnSocketLink`	Ball and socket link. The joint has 3 degrees of freedom of movement and 6 degrees of freedom of placement.
`FixedRootLink`	Fixed system root link.
`ZScrewLink`	This is used as a spacer to provide two extra degrees of freedom to the placement of another link (typically an MDH link).
`Widget`	The widget base class.
`GroundPointsWidget`	Collision detection between a link and the ground.
`PositionSensorWidget`	Sense the absolute position of some point in a link.
`SpringWidget`	A spring between two points in two links.
`SimpleMotorWidget`	A simple linear force joint motor.
`SpatialThrusterWidget`	An arbitrary linear and rotational force.
`JointLimiterWidget`	Spring plus damper limits on joint movement.
`RootLink`	Extra data stored for system root links.

Appendix J

Dyson

J.1 Introduction

Dyson is a software system for dynamic system design and simulation. Dyson has the following components:

- A dynamic network system description language called 'DND'.
- A compiler called '`dnd2exe`' (written in Perl) to convert DND into executable C code.
- A simulation kernel (written in C) to simulate the system.

Dyson is much smaller in scope than professional dynamic modeling packages. It currently lacks the ability to form algebraic loops of stateless variables, which means it can not encompass constraint equations between variables. Dyson is intended for dynamical system *design* rather than *modeling*.

J.2 `dnd2exe`

J.2.1 Introduction

`dnd2exe` is a Perl script that take (as standard input) a DND file and produces (as standard output) C code that defines the following functions:

```
void GetSystemInfo (int *nums, int *numd, int *numu, int *numi,
    char **_statenames[], char **_sdnames[], char **_unitnames[],
    char **_maininputnames[]);
void InitializeStateVariables (double *s, double _stepsize);
void InitializeStepDelayUnits (double *d);
void GetStateDerivatives (double t, double stepsize, double *s,
    double *d, double *x, double *in, double *ds);
void GetStepDelayOutputs (double t, double stepsize, double *s,
    double *d, double *x, double *in, double *d2);
void GetStatelessVariables (double t, double stepsize, double *s,
    double *d, double *in, double *x);
void InitializeSystem (double _stepsize);
void ResetSystem (double _stepsize);
```

J.2.2 Simulation external C functions

The simulation functions do the following:

- `GetSystemInfo` : Returns a number of system parameters to the caller:
  - `int nums` : The number of continuous state variables.
  - `int numd` : The number of step delay units.
  - `int numu` : The number of stateless (unit) variables.
  - `int numi` : The number of inputs to the 'main' module.
  - `char *statenames[]` : An array of size `nums` containing the names of the state variables.
  - `char *sdnames[]` : An array of size `numd` containing the names of the step delay units.
  - `char *unitnames[]` : An array of size `numu` containing the names of the stateless (unit) variables.
  - `char *maininputnames[]` : An array of size `numi` containing the names of the inputs to the 'main' module.
- `InitializeStateVariables` : Set the initial values of the state variable array `s`.
- `InitializeStepDelayUnits` : Set the initial values of the step delay unit array `d`.
- `GetStateDerivatives` : Given the time `t`, the state variable array `s` (of size `nums`), the step delay variable array `d` (of size `numd`), the stateless variable array `x` (of size `numu`), and the input array `in` (of size `numi`), return the state derivatives in the array `ds` (which is of size `nums`).
- `GetStepDelayOutputs` : Given the time `t`, the state variable array `s` (of size `nums`), the step delay variable array `d` (of size `numd`), the stateless variable array `x` (of size `numu`), and the input array `in` (of size `numi`), return the new step outputs of the step delay units in the array `d2` (which is of size `numd`).
- `GetStatelessVariables` : Given the time `t`, the state variable array `s` (of size `nums`), the step delay variable array `d` (of size `numd`) and the input array `in` (of size `numi`), return the stateless variables in the array `x` (which is of size `numu`).
- `void InitializeSystem (double _stepsize)` : Initialize stuff which should only be initialized once, i.e. not every time the simulation is reset.
- `void ResetSystem (double _stepsize)` : Reset some simulation stuff between simulation iterations.

J.2.3 Command line arguments

The command line arguments to `dnd2exe` are

- `-p name` : Add the given name as a prefix to all C external function names.
- `-d` : Produce debugging output as a C comment.
- `-i filename` : Set the name of a file to `#include` at the start of the generated C code.
- `-f` : Produce code that uses floats instead of doubles.

J.3 Dynamic network description (DND) language

J.3.1 Introduction

DND is a simple language for high level description of a modular dynamical systems such as a neural networks. With DND you can:

- Specify state variable equations, both continuous and discrete (such as an integrator or first order filter).
- Specify stateless variable equations (where the variable is updated "instantly" at each time step, i.e. there is no time delay).
- Create sub-modules which can form components of larger systems.
- Group variables together and pass the group between modules.

J.3.2 Grammar (in pseudo-BNF)

The DND file is a sequence of tokens that obey the grammar shown here. Comments are written after a '#' character and continue until the end of line. In this grammar character literals are written in upper case, they stand for the string that is the lower case equivalent. Symbol literals are written as they are, e.g. '+'.

```
file:
        global_definitions

global_definitions:
        global_definition
        global_definitions global_definition

global_definition:
        function_definition
        group_definition
        module_definition

function_definition:
        FUNCTION name integer [+];

group_definition:
        GROUP name { groupname_list }

groupname_list:
        name
        name = number
        groupname_list , name

module_definition:
        MODULE name ( module_arglist_or_null ) { module_body }
```

```
module_arglist_or_null:
        (null)
        module_arglist

module_arglist:
        name
        name = expression
        group_name name
        module_arglist , name

module_body:
        statement
        module_body statement

statement:
        INT name ;
        INT name ( expression , expression ) ;
        SD name ;
        SD name ( expression , expression ) ;
        UNIT name ;
        UNIT name expression ;
        SUMUNIT name ;
        LINK expression -> name ;
        module_name name ( module_args_1_or_null );
        module_name name [ module_args_2_or_null ];
        group_name name ( group_members ) ;
        UNIQUE name ;
        [INITIAL | RESET] function_name ( comma_expression_or_null ) ;

module_args_1_or_null:
        (null)
        module_args_1

module_args_1:
        expression
        module_args_1 , expression

module_args_2_or_null:
        (null)
        module_args_2

module_args_2:
        name = expression
        module_args_2 , name = expression
```

```
group_members:
        name
        group_members , name

expression:
        or_expression

or_expression:
        xor_expression
        or_expression OR xor_expression

xor_expression:
        and_expression
        xor_expression XOR and_expression

and_expression:
        relop_expression
        and_expression AND relop_expression

relop_expression:
        sum_expression
        relop_expression relop sum_expression

sum_expression:
        product_expression
        sum_expression + product_expression
        sum_expression - product_expression

product_expression:
        unary_expression
        product_expression * unary_expression
        product_expression / unary_expression

unary_expression:
        primary_expression
        + unary_expression
        - unary_expression
        NOT unary_expression

primary_expression:
        name
        number
        ( expression )
        function_name ( comma_expression_or_null )

comma_expression_or_null:
```

```
        (null)
        comma_expression

comma_expression
        expression
        comma_expression , expression

name, module_name, group_name:
        Regexp: [A-Za-z_][A-Za-z0-9_.]*

integer:
        Regexp: [0-9]+

number:
        Regexp: [0-9]+([.][0-9]+)?([Ee][+-]?[0-9]+)?

relop:
        <
        >
        <=
        >=
        ==
        !=
```

J.3.3 Expressions

The `and`, `or` and `not` operators work on floating point variables, so they implement the following functions:

- a `and` b = `(a > 0.5) && (b > 0.5)`
- a `or` b = `(a > 0.5) || (b > 0.5)`
- a `xor` b = `(a > 0.5) ^ (b > 0.5)`
- `not` a = `(a <= 0.5)`

The relational operators return 0 or 1.

J.3.4 Statements

This section explains the semantics behind the various syntactical structures shown in the grammar. Later sections will give a fuller explanation of how to use the features of the language.

Global definitions

`function` function-name number-of-parameters [`+`] `;`

Declares that a function (which takes the given number of `double` parameters) has been defined externally (e.g. in the C runtime library). This function can now be used in expressions. If the + is given then the fixed arguments can be followed by any number of variable arguments. In the actual C function call all the arguments are prefixed by an integer which says how many arguments are supplied in the variable section.

`group` groupname { name, name, ... }

Makes 'groupname' a synonym for a group of named variables. This is similar in principle to C's `struct` statement. Default (constant) values can be provided for group elements by saying 'name=number'.

`module` modulename (argument_list) module_body...

Defines a module. The argument list can be empty or it can be a comma separated list of arguments. An argument can be a name followed by an optional '=expression' which gives the default value for the argument. Alternatively the name can be a group name (previously declared with a `group` command) followed by an argument name. See below for how this is interpreted. The module body is a list of statements. Each statement is terminated by a semicolon.

Module statements

`int` name `;` `int` name (derivative_expression, initial_value_expression) `;`

Defines an integrator (state variable) and (optionally) gives expressions for its state derivative and initial value. If no expressions are given then this is a forward declaration, and the integrator must be properly defined later. The initial value expression must evaluate to a constant.

`sd` name `;` `sd` name (next_output_expression, initial_value_expression) `;`

Defines a step delay (discrete state variable) unit and (optionally) gives expressions for its next output and initial value. If no expressions are given then this is a forward declaration, and the step delay unit must be properly defined later. The initial value expression must evaluate to a constant.

`unit` name `;` `unit` name = expression `;`

Defines a stateless variable and (optionally) gives an expression for its output. If no expression is given this is a forward declaration, and the unit must be properly defined later. Algebraic loops in unit expressions are not allowed, so there must not be circular dependencies among units that cannot be resolved without extra equation solving at each iteration.

`sumunit` name type `;`

This defines a unit that can be "linked to". The unit's expression is built using the `link` commands — the unit type determines how the `link` command source expressions are combined to get the unit expression. For sumunits the source expressions are summed together.

module_type module_name (argument_expressions) `;`

This makes an instance of a module 'module_type' called 'module_name'. The module argument names are substituted for the argument expressions given (there must be a one-to-one correspondence). The argument expressions are comma separated.

module_type module_name [argument_assignment_expressions] `;`

This also makes an instance of a module, but here argument assignment expressions can be used, of the form 'name=expression'. The assignment expressions can occur in any order (they do not have to match the declaration order of the module). Module argument names can be repeated, the later assignments will take precedence over the earlier ones (this is a useful mechanism for overriding group arguments). Omitted arguments will take their default values. It is an error if a module argument has neither default value or assignment expression.

group_type group_name (expression_list) ;

This makes an instance of a group. The comma separated expression list gives the value for each element of the group. The group name can now be used as a synonym for the expression list.

`unique` name ;

This declares a value that will be unique for each module that it is used in, e.g. if the **unique** name is instanced three times then the three instances will have the values 0,1,2.

`initial` function_name (arguments) ;

This says that the given function will be executed once, before simulation starts. The arguments should be constant values, but "unique" variables can also be used. As a special case, the `_stepsize` variable can also be used, as it is assumed to remain constant.

`reset` function_name (arguments) ;

This says that the given function will be executed whenever the simulation is reset. The arguments should be constant values, but "unique" variables can also be used. As a special case, the `_stepsize` variable can also be used, as it is assumed to remain constant.

J.3.5 Modules

A module is a subsystem which can be duplicated multiple times. Within the module new integrators and units can be defined. By default these elements will have local scope, that is their names will not be accessible outside the module. A module has named inputs that can be referenced by expressions within the module.

Module instantiation is similar to macro expansion. When a module is instantiated the module body is substituted for the module command, with the following transformation:

- All module variables are prefixed by the module name followed by a dot.
- All module inputs are substituted for the expressions supplied in the module command.

For example:

```
module foo (a,b,c) {
  int tox (a+b+2, 2-c);
  unit output = b*c;
}
module bar (q) {
  unit r = 2;
  foo dof (q,r*4,1)
  unit pod = dof.tox + dof;
}
```

...makes the module 'bar' equivalent to...

```
module bar (q) {
  unit r = 2;
  int dof.tox (q+(r*4)+2, 2-1);
  unit dof.output = (r*4)*1;
  unit pod = dof.tox + dof.output;
}
```

A module called 'main' must be defined. The main module is the one that is actually used for simulation. Its inputs are the global inputs to the system and must be supplied by some external agent.

A module can be forward declared by instantiating it without supplying any arguments, e.g.

```
bar b;
unit x = b.pod;
bar b (123);
```

J.3.6 Expressions and variables

All expressions can only refer to the following variables:

- State variables (integrators) or stateless (unit) variables defined locally within the module. Variables must be defined before they are used. As a special case, an integrator expression can refer to itself.
- Module input variables.
- Integrators and units inside module instances, using the dotted qualifying notation. Referring to a module name itself will refer to a variable called 'name.output'.
- The special variable '`t`' can be used for the current time.
- The special variable '`_stepsize`' can be used for the simulation step size (assuming that it is always the same).

All variables have module local scope.

J.3.7 Groups

Groups allow you to deal with large collections of variables/expressions easily. Groups are implemented using a text-expansion mechanism (a bit similar to macro expansion only smarter). Note that defining a group merely makes the group name a synonym for the grouped variables/expressions, it does not define any new variables.

Suppose we have a group 'g':

```
group g {a,b=2}
```

If we use the group as a module argument like this

```
module mymod (p, g foo, q) {...
```

it is equivalent to writing

```
module mymod (p, foo.a, foo.b=2, q) {
  g foo (foo.a, foo.b);
  ...
```

Note that the group arguments get their default values from the `group` declaration (this is the only way that group arguments can have defaults). The group is re-instantiated inside the module so it can still be referred to collectively. If we then pass a group to '`mymod`' inside another module like this

```
g bar (zog,123);
mymod nog [p=1, foo=bar, q=2];
```

it is equivalent to writing

```
mymod nog [p=1, foo.a=zog, foo.b=123, q=2];
```

Now here is a tricky point. The type of '`foo`' in module '`mymod`' is not stored (we just store the arguments `foo.a` etc) so when the group `bar` was passed to the module, *its* type was used to generate the expanded argument list. When this group expansion is done, arguments that are not one of the module's named inputs are omitted. This means that incompatible groups can be passed between modules, as long as enough default argument values exist to patch up the gaps.

When a module is used inside another module, all of its groups are available for reference.

J.3.8 Constant folding

There is not provision to explicitly declare constants, but you can get the same effect by saying

```
unit my_constant_pi = 3.14;
```

A constant folding mechanism will find all references to (and expressions made up of) constant units and replace them by the constant values.

Appendix K

Real time control system

K.1 Introduction

All hardware control experiments described in this thesis were performed using the RTC system developed by the author. RTC was written in C/C++ and runs under the Linux operating system. Its structure is shown in figure K.1. It consists of a number of Linux kernel modules, kernel modifications, and an X-windows supervisor application. The kernel modules perform the control tasks: A/D sampling, computing the control action and D/A output. The supervisor application graphs A/D sensor values and allows the user to adjust controller parameters online. The interface to the external world is through a "Keithley Data Acquisition and Control" (KDAC) model 575 A/D and D/A system.

Developing kernel modules can be difficult, and using them is inconvenient, but they are able to achieve much more accurate controller scheduling, and also direct access to memory and the I/O space is easier. The controller sampling function is interrupt driven, it is not a standard Linux process because they are unable to achieve reliable real-time scheduling.

Four controller modules were developed for driving different hardware. They are: `crane` which drives a model gantry crane, `invp` which balances an inverted pendulum on a cart, `robot` which steers a toy robot along a track, and `thermal` which controls a model heat-transfer process. The module `invp` is shown in figure K.1.

K.2 Supervisor application

The supervisor application, shown in figure K.2, runs under X-Windows. It communicates with the RTC kernel modules and allows the user to

- Start/stop the controller at a given sampling frequency.
- Display a moving graph of selected sensor values at a user specified magnification.
- Modify controller parameters (such as linear controller gains or FOX learning rates) with sliders, or load them from files.
- Load, save and reset FOX weight tables.
- Start and stop FOX training.

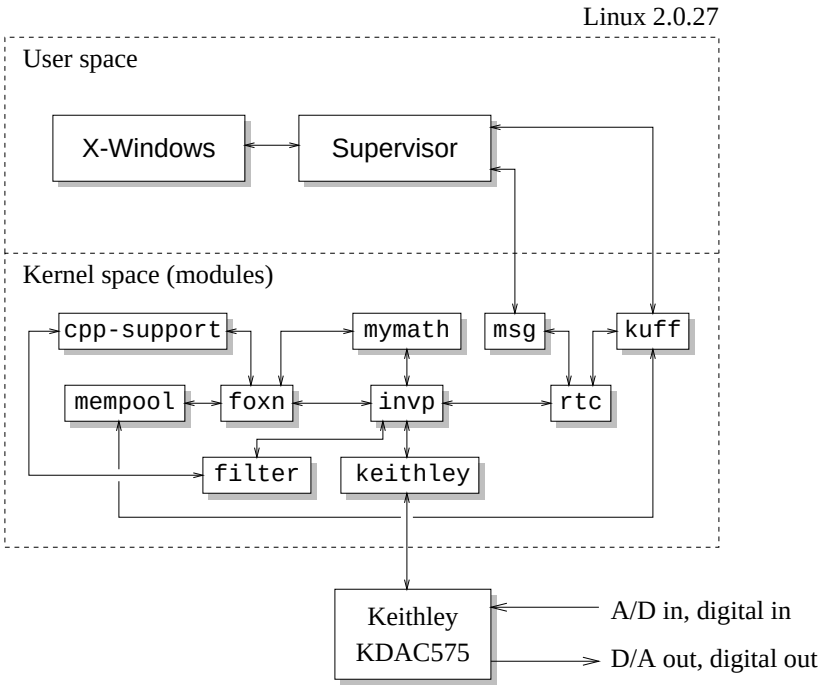


Figure K.1: *The real-time control (RTC) system.*

- Log sensor data to files.
- Select the controller mode (e.g. linear PD or FOX-in-the-loop).

K.3 Kernel modification: `mempool`

`mempool` is a patch to the 2.0.27 Linux kernel which allows a large amount of physical memory to be reserved at boot-time. Pages in this memory pool can be allocated and deallocated by other drivers and kernel modules. It is useful for getting around the size limitation of `kmalloc()`. Also, this memory is safe to `remap_page_range()` into user space, e.g., for `mmap()`. This patch is based on the `bigphysarea` patch by M. Welsh (`mdw@cs.cornell.edu`).

RTC uses `mempool` to allocate large amounts of weight storage and buffers for the FOX algorithm. It is also used by the `kuff` module for allocating buffers that can be shared with users-space processes. To use it, specify the boot option

```
mempool=<number of pages>
```

to specify the number of 4k pages to reserve. If this option is not used then no pages will be reserved. Note that at least one extra page will be reserved for use by a page mapping table. This also adds a file into the `/proc` filesystem called `mempool`, which contains some status information about the memory pool, and also contains a representation of the page map table. The following kernel files are added/modified by the patch:

```
/usr/src/linux/include/linux/mempool.h
/usr/src/linux/mm/mempool.c
```

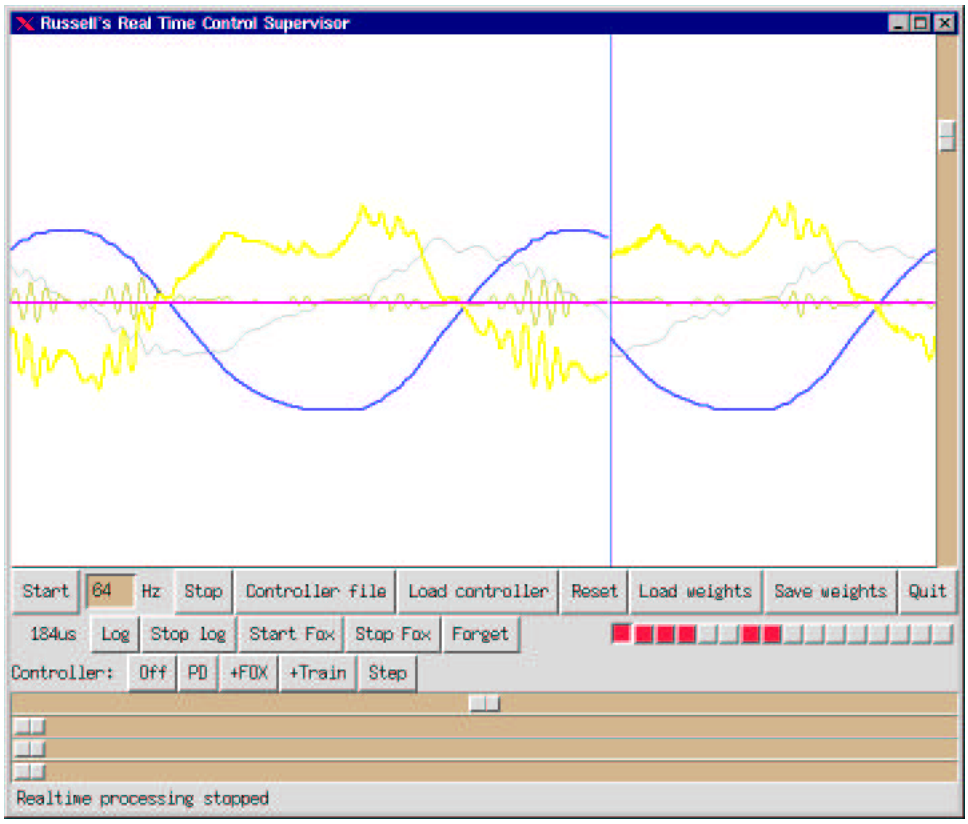


Figure K.2: *The RTC supervisor application.*

K.4 Kernel modules

`invp`, `crane`, `robot`, `thermal` — These modules implement the actual controller algorithms.

`rtc` — The real time control core. This allows a control function to be registered by another kernel module. Communications channels are provided between this module and a user space process (USP). The USP can command the module to start/stop calling the control function at some specified sampling frequency, and it can set controller parameters. Data returned by the controller function is packaged and returned to the USP, along with timing statistics. The PC real-time clock interrupt is used to invoke the control function. The interrupt rate can be set to powers of 2 between 2 Hz and 8192 Hz.

`foxn` — This provides a C++ class called `FoxN` that implements the FOX algorithm.

`keithley` — This provides an interface to the "Keithley Data Acquisition and Control" (KDAC) model 575 A/D and D/A system. Functions are provided to configure/read/write to the A/D, D/A, and digital I/O channels.

`kuff` — This is the **k**ernel-**u**ser **f**ast **f**ifo channel. A fast fifo is a device for one-way communication between the kernel and a user-space process. The fifo is "fast" because the user process gets to manipulate the fifo buffer memory directly. Every `kuff` device has a corresponding file `/dev/kuff[1-4]`. The user space process can not `read()` and `write()` data directly from/to these files. Instead a 4-byte structure is `read()` which gives a length value, which is used to `mmap()` the fifo buffer into the user address space. Note that `select()` can also be called on the device.

`msg` — The `msg` device is a very simple two-way message communications path between the kernel and a user-space process (USP). From USP to kernel: The kernel can set handlers on the `/dev/msg` devices. When the USP `write()`s data to a `/dev/msg` it is sent directly to the handler (if any). From kernel to USP: The kernel can set a message buffer to be read by the USP. When the USP `read()`s data from a `/dev/msg` device, the message buffer is returned (if any). Reads and writes will never block, and will always be atomic.

`cpp-support` — This provides low level support for C++ object files in the kernel. It provides the built-in `new` and `delete` functions.

`filter` — This provides C++ FIR and IIR filter classes for filtering A/D data. Pre-initialized 3rd and 5th order Butterworth filters are provided (with cutoffs of 0.05 and 0.15 times the sampling frequency, respectively).

`mymath` This module provided the following math library functions (which are otherwise unavailable to the kernel): `sin`, `cos`, `exp`, `log`, `ceil`, `rint`, `fabs`, `pow`.

Bibliography

[1] George Adelman, editor. *Encyclopedia of Neuroscience*, volume I. Birkhäuser, 1987.

[2] J.S. Albus. Data storage in the cerebellar model articulation controller (CMAC). *Transactions of the ASME: Journal of Dynamic Systems, Measurement, and Control*, pages 228–233, September 1975.

[3] J.S. Albus. A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Transactions of the ASME: Journal of Dynamic Systems, Measurement, and Control*, pages 220–227, September 1975.

[4] Jennifer S. Altman and Jenny Kien. New models for motor control. *Neural Computation*, 1:173–183, 1989.

[5] Shun-ichi Amari. Mathematical foundations of neurocomputing. *Proceedings of the IEEE*, 78(9), September 1990.

[6] Shun-ichi Amari and Akikazu Takeuchi. Mathematical theory on formation of category detecting nerve cells. *Biological Cybernetics*, 29:127–136, 1978.

[7] H.C. Andersen, F.C. Teng, and A.C. Tsoi. Single net indirect learning architecture. *IEEE Transactions on Neural Networks*, 5(6):1003–1005, November 1994.

[8] Peter J. Angeline, Gregory M. Saunders, and Jordan B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1), January 1994.

[9] Robert J. Baron. *The cerebral computer : an introduction to the computational structure of the human brain*. L. Erlbaum Associates, 1987.

[10] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuron-like adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13(5):834–846, 1983.

[11] Ulrich Bässler. The walking-(and searching-) pattern generator of stick insects, a modular system composed of reflex chains and edogenous oscillators. *Biological Cybernetics*, 69:305–317, 1993.

[12] Françoise Beaufays and Eric A. Wan. Relating real-time backpropagation and backpropagation-through-time: An application of flow graph interreciprocity. *Neural Computation*, 6:296–306, 1994.

[13] Randall D. Beer. *Intelligence as Adaptive Behavior: An Experiment in Computational Neuroethology*. Academic Press, 1990.

[14] Randall D. Beer, Hillel J. Chiel, Roger D. Quinn, Kenneth S. Espenschied, and Patrick Larsson. A distributed neural network architecture for hexapod robot locomotion. *Neural Computation*, 4:356–365, 1992.

[15] Randall D. Beer, Hillel J. Chiel, and Leon S. Sterling. An artificial insect. *American Scientist*, 79:444–452, 1991.

[16] Geoffrey Black. Control of an overhead gantry crane. Technical Report 1997-EL07, University of Auckland, New Zealand, 1997.

[17] G. N. Boone and J. K. Hodgins. Slipping and tripping reflexes for bipedal robots. *Autonomous Robots, Special Issue on Biped Locomotion of Autonomous Robots*, 4(3), 1997.

[18] Gary Boone and Jessica Hodgins. Walking and running machines. In *The MIT Encyclopedia of the Cognitive Sciences*. MIT Press, 1998.

[19] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1), March 1986.

[20] Rodney A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. *Neural Computation*, 1:253–262, 1989.

[21] Martin Brown and Chris Harris, editors. *Neurofuzzy adaptive modeling and control*. Prentice Hall, New York, 1994.

[22] Martin Brown, Christopher J. Harris, and Patrick C. Parks. The interpolation capabilities of the binary CMAC. *Neural Networks*, 6:429–440, 1993.

[23] Dean V. Buonomano and Michael D. Mauk. Neural network model of the cerebellum: Temporal discrimination and the timing of motor responses. *Neural Computation*, 6:38–55, 1994.

[24] Thierry Catfolis. A method for improving the real-time recurrent learning algorithm. *Neural Networks*, 6:807–821, 1993.

[25] M.M. Chawla and S.R. Sharma. Families of three-stage third order Runge-Kutta-Nyström methods for $y'' = f(x, y, y')$. *Journal of the Australian mathematical society, Series B*, 26:375–386, 1985.

[26] J.J. Collins and S.A. Richmond. Hard-wired central pattern generators for quadrupedal locomotion. *Biological Cybernetics*, 71:375–385, 1994.

[27] J.J. Collins and Ian Stewart. Hexapodal gaits and coupled nonlinear oscillator models. *Biological Cybernetics*, 68:287–298, 1993.

[28] Christopher I. Connolly and J. Brian Burns. A state-space striatal model. In J. Houk, editor, *Models of information processing in the basal ganglia*. MIT Press, October 1993.

[29] Lara S. Crawford and S. Shankar Sastry. Biological motor control approaches for a planar diver. In *Proceedings of the 34th IEEE Conference on Decision and Control*, pages 3881–3886, December 1995.

[30] Lara S. Crawford and S. Shankar Sastry. Learning controllers for complex behavioral systems. Technical report, ERL Memo Number M96/73, U.C. Berkeley, 1996.

[31] H. Cruse, D.E. Brunn, and Ch. Bartling *et al*. Walking: A complex behavior controlled by simple networks. *Adaptive Behavior*, 3(4):385–418, 1995.

[32] Örjan Ekeberg, Anders Lansner, and Sten Grillner. The neural control of fish swimming studied through numerical simulations. *Adaptive Behavior*, 3(4):363–384, 1995.

[33] Martin Eldracher, Alexander Staller, and René Pompl. Adaptive encoding strongly improves function approximation with CMAC. *Neural Computation*, 9:403–417, 1997.

[34] Roy Featherstone. *Robot Dynamics Algorithms*. Kluwer Academic Publishers, 1987.

[35] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice, 2nd Edition*. Addison-Wesley, 1990.

[36] Mitsuo Gen and Runwei Cheng. *Genetic Algorithms and Engineering Design*. Wiley Interscience, 1997.

[37] Konrad Gesner. *Curious woodcuts of fanciful and real beasts; a selection of 190 sixteenth-century woodcuts from Gesner's and Topsell's natural histories*. Dover Publications, 1971.

[38] Hiroaki Gomi and Mitsuo Kawato. Neural network control for a closed loop system using feedback-error-learning. *Neural Networks*, 6:933–946, 1993.

[39] V. Gullapalli, J.A. Franklin, and H. Benbrahim. Acquiring robot skills via reinforcement learning. *IEEE Control Systems Special Issue on Robotics: Capturing Natural Motion*, 4(1):13–24, February 1994.

[40] Arthur C. Guyton. *Structure and Function of the Nervous System*. W.B. Saunders Company, 1972.

[41] Bridget E. Hallam, Janet R.P. Halperin, and John C.T. Hallam. An ethological model for implementation in mobile robots. *Adaptive Behavior*, 3(1):51–79, 1994.

[42] David A. Handelman and Stephen H. Lane. Fast sensorimotor skill acquisition based on rule-based training of neural networks. In George A. Bekey and Kenneth Y. Goldberg, editors, *Neural networks in robotics*, pages 255–270. Kluwer Academic Publishers, 1993.

[43] David A. Handleman, Stephen H. Lane, and Jack J. Gelfand. Integrating neural networks and knowledge-based systems for intelligent robotic control. *IEEE Control Systems Magazine*, April 1990.

[44] Donald Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Wiley (New York), 1949.

[45] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computation*. Addison Weseley, 1991.

[46] Shinya Hosogi, Nobuo Watanabe, and Minoru Sekiguchi. A neural network model of the cerebellum performing dynamic control of a robotic manipulator by learning. *Fujitsu Science and Technology Journal*, 29(3):201–208, September 1993.

[47] Dean Hougen, John Fischer, and Deva Johnam. A neural network pole balancer that learn and operates on a real robot in real time. In *Proceedings of the MLC-COLT workshop on robot learning*, pages 73–80, 1994.

[48] Yendo Hu and Ronald D. Fellman. An implementation efficient learning algorithm for adaptive control using associative content addressable memory. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(4), 1995.

[49] Don R. Hush and Bill G. Horne. Progress in supervised neural networks. *IEEE Signal Processing Magazine*, January 1993.

[50] Ken ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks*, 6:801–806, 1993.

[51] Petros A. Ioannou. *Robust adaptive control*. Prentice-Hall, 1996.

[52] Hisao Ishibuchi, Ryosuke Fujioka, and Hideo Tanaka. Neural networks that learn from fuzzy if-then rules. *IEEE Transactions on fuzzy systems*, 1(2), May 1993.

[53] Masao Ito. *The Cerebellum and Neural Control*. Raven Press, 1984.

[54] Henry C. Jen, Zhuang Tian, Russell L. Smith, and George G. Coghill. Hardware implementation of a CMAC neural network using PLDs. In *Proceedings of the 8th Australian Conference on Neural Networks*, pages 40–44, June 1997.

[55] Leslie Pack Kaebling and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4(Jan-Jun):237–285, 1996.

[56] Shuuji Kajita and Kazuo Tani. Experimental study of biped dynamic walking. *IEEE Control Systems Magazine*, February 1996.

[57] K. Th. Kalveram. Controlling the dynamics of a two-joined arm by central patterning and reflex-like processing. *Biological Cybernetics*, 65:65–71, 1991.

[58] Eric R. Kandel and Robert D. Hawkins. The biological basis of learning and individuality. *Scientific American*, September 1992.

[59] Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell, editors. *Principles of Neural Science, 3rd edition*. Appleton & Lange, 1991.

[60] M. Kawato, Kazunori Furukawa, and R. Suzuki. A hierarchical neural-network model for control and learning of voluntary movement. *Biological Cybernetics*, 57:169–185, 1987.

[61] A.H. Klopf. Classical conditioning: Phenomena predicted by drive-reinforcement model of neural function. In J.H. Byrne and W.O. Berry, editors, *Neural Models of Plasticity: Experimental and Theoretical Approaches*, chapter 6, pages 94–103. Academic Press, 1989.

[62] Stephen H. Lane, David A. Handelman, and Jack J. Gelfand. Modulation of robotic motor synergies using reinforcement learning optimization. In George A. Bekey and Kenneth Y. Goldberg, editors, *Neural networks in robotics*, pages 521–538. Kluwer Academic Publishers, 1993.

[63] Stephen H. Lane, David A. Handleman, and Jack J. Gelfand. Theory and development of higher-order CMAC neural networks. *IEEE Control Systems Magazine*, April 1992.

[64] Joseph Laszlo, Michiel van de Panne, and Eugene Fiume. Limit cycle control and its application to the animation of balancing and walking. In *Computer Graphics Proceedings, Annual Conference Series*, pages 155–162, 1996.

[65] Mark L. Latash. *Control of Human Movement*. Human Kinetics Publishers, Champaign, Illinois, 1993.

[66] M. Anthony Lewis, Andrew H. Fagg, and George A. Bekey. Genetic algorithms for gait synthesis in a hexapod robot. In Yuan F. Zheng, editor, *Recent trends in mobile robotics*, chapter 11, pages 317–331. World Scientific, 1993.

[67] Chun-Shin Lin and Hyongsuk Kim. CMAC-based adaptive critic self- learning control. *IEEE Transactions on Neural Networks*, 2(5), 1991.

[68] Chun-Shin Lin and Hyongsuk Kim. Selection of learning parameters for CMAC-based adaptive critic learning. *IEEE Transactions on Neural Networks*, 6(3), 1995.

[69] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

[70] Yi Lin and Shin-Min Song. A CMAC neural network for the kinematic control of walking machine. In George A. Bekey and Kenneth Y. Goldberg, editors, *Neural networks in robotics*, pages 29–44. Kluwer Academic Publishers, 1993.

[71] Richard P. Lippman. An introduction to computing with neural nets. *IEEE ASSP Magazine*, April 1987.

[72] Rainer Malaka and Martin Hammer. Real-time models of classical conditioning. In *Proceedings of the International Conference on Neural Networks (ICNN'96), Washington*, volume 2, pages 768–773. IEEE Press, Piscataway, N.J., 1996.

[73] Rainer Malaka, Reiner Lange, and Martin Hammer. A constant prediction model for classical conditioning. In N. Elser and R. Menzel, editors, *Learning and Memory, Proceedings of the 23rd Göttingen Neurobiology conference*, volume 1, page 57. Thieme-Verlag, Stuttgart, New York, 1995.

[74] Larry Matthies *et al*. Mars microrover navigation: Performance evaluation and enhancement. *Proceedings of the IEEE/RSJ International Conference on Robots and Systems (IROS)*, August 1995.

[75] B.J. McFadyen, D.A. Winter, and F. Allard. Simulated control of unilateral, anticipatory locomotor adjustments during obstructed gait. *Biological Cybernetics*, 72:151–160, 1994.

[76] Scott McMillan. *Computational Dynamics for Robotic Systems on Land and Under Water*. PhD thesis, Ohio State University, September 1994.

[77] Scott McMillan, David E. Orin, and Robert B. McGhee. Efficient dynamic simulation of an underwater vehicle with a robotic manipulator. *IEEE Transactions on Systems, Man and Cybernetics*, 25(8):1194–1206, August 1995.

[78] Scott McMillan, P. Sadayappan, and David E. Orin. Efficient dynamic simulation of multiple manipulator systems with singular configurations. *IEEE Transactions on Systems, Man and Cybernetics*, 24(2):306–313, February 1994.

[79] Scott McMillan, P. Sadayappan, and David E. Orin. Parallel dynamic simulation of multiple manipulator systems: Temporal vs spatial methods. *IEEE Transactions on Systems, Man and Cybernetics*, 24(7):982–989, July 1994.

[80] Lisa Meeden, Gary McGraw, and Douglas Blank. Emergent control and planning in an autonomous vehicle. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, 1994.

[81] José del R. Millán. Learning efficient reactive behavioral sequences from basic reflexes in a goal-directed autonomous robot. In *From Animals to Animats: Third International Conference on Simulation of Adaptive Behavior*, August 1994.

[82] Paul I. Miller. Recurrent neural networks and motor programs. In *Fifth European Symposium On Artificial Neural Networks*, April 1997.

[83] W. Thomas Miller. Learning dynamic balance of a biped walking robot. In *ICANN 1994*, pages 2771–2777, 1994.

[84] W. Thomas Miller. Real-time neural network control of a biped walking robot. *IEEE Control Systems Magazine*, pages 41–48, February 1994.

[85] W. Thomas Miller, Filson Glanz, and L. Gordon Kraft. CMAC: An associative neural network alternative to backpropagation. *Proceedings of the IEEE*, 78(10), 1990.

[86] W. Thomas Miller, Richard Sutton, and Paul Werbos, editors. *Neural Networks for Control*. MIT Press, Cambridge, Massachusetts, 1990.

[87] H. Miyamoto, M. Kawato, T. Setoyama, and R. Suzuki. Feedback-error-learning neural network for trajectory control of a robotic manipulator. *Neural Networks*, 1:251–265, 1988.

[88] J.W. Moore and D.E.J. Blazis. Cerebellar implementation of a computational model of classical conditioning. In *The olivocerebellar system in motor control*. Springer-Verlag, 1989.

[89] Kumpati S. Narendra and Kannan Parthasarathy. Identification and control of dynamic systems using neural networks. *IEEE Transactions on Neural Networks*, 1(1):4–27, March 1990.

[90] Jun Nishii, Yoji Uno, and Ryoji Suzuki. Mathematical models for the swimming pattern of a lamprey: I. Analysis of collective oscillators with time-delayed interaction and multiple coupling. *Biological Cybernetics*, 72:1–9, 1994.

[91] Jun Nishii, Yoji Uno, and Ryoji Suzuki. Mathematical models for the swimming pattern of a lamprey: II. Control of the central pattern generator by the brainstem. *Biological Cybernetics*, 72:11–18, 1994.

[92] Hiroshi Ohno, Toshihiko Suzuki, Keiji Aoki, Arata Takahasi, and Gunji Sugimoto. Neural network control for automatic braking control system. *Neural Networks*, 7(8):1303–1312, 1994.

[93] Satinder Pannu, H. Kazerooni, Gergory Becker, and Andrew Packard. μ-Synthesis control for a walking robot. *IEEE Control Systems Magazine*, February 1996.

[94] P.C. Parks and J. Militzer. Improved allocation of weights for associative memory storage in learning control systems. *Proceedings of the 1st IFAC symposium on design methods for control systems, in Zürich*, pages 777–782, 1991.

[95] Barak A. Pearlmutter. Learning state space trajectories in recurrent neural networks. *Proceedings of the IJCNN*, 2:365–372, 1989.

[96] Barak A. Pearlmutter. Dynamic recurrent neural networks. *Technical Report CMU-CS-90-196, Carnegie Mellon University, School of Computer Science*, 1990.

[97] Barak A. Pearlmutter. Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural Networks*, 6(5), September 1995.

[98] P. Peretto. *An introduction to the modeling of neural networks*. Cambridge University Press, 1992.

[99] Fernando J. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59(19), 1987.

[100] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing, 2nd edition*. Cambridge University Press, 1992.

[101] Karl H. Pribram. *Languages of the brain; experimental paradoxes and principles in neuropsychology*. Prentice-Hall, 1971.

[102] Demetri Psaltis, Athanasios Sideris, and Alan A. Yamamura. A multilayered neural network controller. *IEEE Control Systems Magazine*, April 1988.

[103] M. H. Raibert. *Legged Robots That Balance*. MIT Press, 1986.

[104] Marc H. Raibert, Benjamin Brown, and Seshashayee S. Murthy. Machines that walk. In M. Brady *et al*, editor, *NATO ASI Series: Robotics and artificial intelligence*, volume F11, pages 345–364. Springer-Verlag, 1984.

[105] George N. Reeke, Olaf Sporns, and Gerald M. Edelman. Synthetic neural modeling: The "Darwin" series of recognition automata. *Proceedings of the IEEE*, 78(9):1498–1530, September 1990.

[106] P. Gonzalez de Santos, M.A. Armada, and M.A. Jimenez. Walking machines: Initial testbeds, first industrial applications and new research. *IEE Computing and control engineering journal*, October 1997.

[107] Michael A. Sartori and Panos J. Antsaklis. Implementations of learning control systems using neural networks. *IEEE Control Systems Magazine*, April 1992.

[108] Wolfram H. Schiffmann and Willi Geffers. Adaptive control of dynamic systems by back propagation networks. *Neural Networks*, 6:517–524, 1993.

[109] Robert Sedgewick. *Algorithms*, chapter 16. Addison Wesley, 1988.

[110] Melani Shoemaker and Blake Hannaford. A study and model of the role of the Renshaw cell in regulating the transient firing rate of the motorneuron. *Biological Cybernetics*, 71:251–262, 1994.

[111] Karl Sims. Evolving 3D morphology and behavior by competition. *Artificial Life IV Proceedings*, pages 28–39, 1994.

[112] Karl Sims. Evolving virtual creatures. *SIGGRAPH'94 Proceedings*, pages 15–22, 1994.

[113] Russell L. Smith. Behavioral control of a robot arm. *Proceedings of the second New Zealand international conference on Artificial Neural Networks and Expert Systems (ANNES)*, pages 346–349, 1995.

[114] Russell L. Smith. A simple method for constructing and evaluating chain-rule propagation algorithms. *Proceedings of the ANNES conference, Dunedin, New Zealand*, pages 38–41, 1995.

[115] Russell L. Smith. An autonomous robot controller with learned behavior. *The Australian Journal of Intelligent Information Processing Systems*, 3(2), Winter 1996.

[116] Steve Stitt and Yuan F. Zheng. Distal learning applied to biped robots. In Yuan F. Zheng, editor, *Recent trends in mobile robotics*, chapter 12, pages 333–358. World Scientific, 1993.

[117] R.S. Sutton. Learning to predict by methods of temporal differences. *Machine Learning*, 9(3):9–44, 1988.

[118] R.S. Sutton and A.G. Barto. Toward a modern theory of adaptive networks: Expectation and prediction. *Psychological Review*, 85:135–170, 1981.

[119] R.S. Sutton and A.G. Barto. Time-derivative models of pavlovian reinforcement. In A. Gabriel and J. Moore, editors, *Learning and computational neuroscience: Foundations and adaptive networks*, chapter 12, pages 497–535. MIT Press, 1990.

[120] G. Taga, Y. Yamaguchi, and H. Shimizu. Self-organized control of bipedal locomotion by neural oscillators in unpredictable environment. *Biological Cybernetics*, 65:147–159, 1991.

[121] Steven L. Tanimoto. *The Elements of Artificial Intelligence Using Common Lisp, 2nd Edition*. Computer Science Press, 1995.

[122] David E. Thompson and Sunggyu Kwon. Neighborhood sequential and random training techniques for CMAC. *IEEE transactions on neural networks*, 6(1), January 1995.

[123] H. Tolle, S. Gehlen, and M. Schmitt. On interpolating memories for learning control. In Kenneth Hunt, George Irwin, and Kevin Warwick, editors, *Neural Network Engineering in Dynamic Control Systems*, pages 127–152. Springer, 1995.

[124] K.P. Unnikrishnan and K.P. Venugopal. Alopex: A correlation-based learning algorithm for feedforward and recurrent neural networks. *Neural Computation*, 6:469–490, 1994.

[125] K.P. Venugopal, A.S. Pandya, and R. Sudhakar. A recurrent neural network controller and learning algorithm for the on-line learning control of autonomous underwater vehicles. *Neural Networks*, 7(5):833–846, 1994.

[126] A.R. Wager. SOP: A model of automatic memory processing in animal behavior. In N.E. Spear and R.R. Miller, editors, *Information processing in animals: Memory mechanisms*, volume 85, chapter 1, pages 5–44. Erlbaum, New Jersey, 1981.

[127] Jeffrey M. Wendlandt and S. Shankar Sastry. Recursive workspace control of multibody systems: A planar biped example. In *Proceedings of the 35th IEEE Conference on Decision and Control*, December 1996.

[128] Paul J. Werbos. An overview of neural networks for control. *IEEE Control Systems Magazine*, January 1991.

[129] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[130] Ronald J. Williams and David Zipser. Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1(1), 1989.

[131] J. Yuh. Learning control for underwater robotic vehicles. *IEEE Control Systems Magazine*, pages 39–45, April 1994.

Colophon

This thesis was originally typeset using LaTeX 2_ε in 11 point Computer Modern Roman. The `epsfig`, `xspace`, `color`, `tabularx` and `longtable` packages were used. The figures were drawn using `xfig` 3.2, except for figure 2.4 and figure 2.12 which are from [59], and figure 2.1 and figure 2.9 which are derived from Corel Draw™ clip-art. The graphs were generated with Matlab 4.2c. The cover art is a 16th century woodcut of a fox from the Dover Pictorial Archive, published in [37].

The online PDF version of this thesis was typeset using PDF-LaTeX, using the `graphicx` and `hyperref` packages in addition to the ones above. It uses a Times Roman font on 8.5×11 inch pages—thus the page numbers do not correspond to the original version.

The Road goes ever on and on
Out from the door where it began.
Now far ahead the Road has gone,
Let others follow it who can!
Let them a journey new begin,
But I at last with weary feet
Will turn towards the lighted inn,
My evening-rest and sleep to meet.

—J.R.R. Tolkien