UNIVERSITY OF SOUTHAMPTON

Parallel Processing Tools in Adaptive and Self Tuning Control

David Matthew Brown

Doctor of Philosophy

January 2000

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING & APPLIED SCIENCE
SCHOOL OF ENGINEERING SCIENCES

Doctor of Philosophy

Parallel Processing Tools in Adaptive and Self Tuning Control

by David Matthew Brown

The accuracy and effectiveness of a control system depends greatly on the sampling rate. Along with each individual plant comes a practical minimum sampling period. If the accuracy of the control of a plant falls below certain criteria, two options are available: either a more complex plant model can be chosen so that it more accurately represents the plant trajectory; or the sampling rate can be increased. However, a more complex model is only an option if the fault lies in the model; and a faster sampling rate is only an option if the computational overheads do not dictate the minimum length of the sampling period. Most control systems will be working at the absolute limit with the sampling period dictated entirely by how long it takes to calculate a set of control inputs from the sampled input and output or state information.

Real-time solutions provide an alternative, through parallel processing, for increased accuracy of control. By finding solutions to run existing algorithms faster, more complex plant models may be implemented and/or sampling rates may be increased.

This thesis develops methods for parallelising existing adaptive control systems before introducing novel solutions to the adaptive control of linear and nonlinear plants through the use of multiple model switching schemes. Formerly non-implementable due to the computational intensity involved, these novel methods are made practical by the high degree of parallelisation that is possible in their algorithms and the much faster calculations that are therefore possible through the use of parallel processing. The parallel concepts of speed-up and scalability are introduced and used for evaluation purposes throughout. Wherever relevant, the parallel results are directly compared against the sequential ones.

Acknowledgements

I would like to thank the following people to whom I am forever grateful:

My supervisors Professor Eric Rogers and Dr Owen Tutty for their help and advice throughout my Ph.D.
My Parents for their financial support, without which this thesis might never have been finished.
Rich, Keith and Jules for the 'occasional' pint when I was really broke.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Adaptive Control

The concept of adaptive control was originally due to the requirement for control systems to emulate nature by being able to adapt to their environments. Since adaptive control is a vague term, the discipline covered a great diversity of areas in control. As a result there have been a number of attempts to classify distinct areas (Aseltine *et al.* 1958; Levin 1958; Jacobs 1964; Aström 1983). The two main areas arising from this refinement are the *Self-Tuning Regulator (STR)* and *Model Reference Adaptive Systems (MRAS)*. STRs owe their past to stochastic systems research and their parameter tuning methods come from statistical analysis; the MRAS come from an engineering background and make use of gradient descent methods such as the MIT Rule and Lyapunov method (see for example Harris and Billings (1985)). However, as far as this thesis is concerned the two fields are synonymous and no great distinction is drawn between them.

The applications of computers to control are limited by the raw processing power available. This is reflected in table 1.1 which gives an idea of the areas

Date	Gas, chemical petroleum, cement	Power	Metals	Misc.	Total
May 1961	16	11	10	0	37
May 1962	45	66	23	23	159
Sept 1963	92	117	55	76	340
Aug 1965	212	203	144	236	795
Sept 1966	336	289	242	485	1351
Mar 1967	386	324	260	601	1571
Jul 1968	–	–	–	–	2890

Table 1.1: Number of computer controlled processes (Wismer and Wells 1972)

of control that computers were first applied to. It is no accident that the list is largely made up of chemical and process industries. These applications were possible because of the slow process dynamics involved in the processes to be controlled. The shear size of computers in the early years also severely restricted their use in portable applications, such as missile control or on board aircraft. The final consideration was the cost of computers which made them practical only in industries where improvements in control could justify the large initial expense (for example by increasing the life of catalysts in a chemical process). The increase in the number of applications over the time covered in the table (less than a decade) gives an indication of the success of these early computer control based schemes.

With Moore's law stating that processing power approximately doubles every eighteen months (Takeda *et al.* 1999), computational speed requirements are not as big a problem nowadays. However, top of the range computers are still expensive and cheaper solutions can often be found through parallel processing involving a small number of low cost commodity processors. Ultimately, an upper sequential processor speed will be reached, and parallel processing solutions will offer the main route to increased processing power beyond this limit. Therefore, deriving parallel processing models for control schemes offers great future advantages.

Early examples of parallel processing applications in control came through so called systolic architectures (described further in chapter 4) where a large number

of simple processors (cells) are linked together in pipelines for the purpose of performing recursive calculations on a large amount of data. Common applications were in the computation of difference equation models (for example see (Kwan 1987; Lin 1986)) and Recursive Least Squares (for example see (Gaston *et al.* 1994; McWhirter and Proudler 1994)). Chisci and Zappa (1993) summarises the objectives of the parallelisation of adaptive control systems as the achievement of:

1. A sampling rate as fast as possible.
2. The shortest possible computation delay between sampling and the generation of the control signal.
3. A parameter tuning rate as fast as possible.
4. The shortest possible delay between sampling and the generation of the regulator parameters.

Of course, points one and two are a universal set of goals for control systems. Most systolic architectures for adaptive control concentrate on either the parallelisation of the tuning method or the computation of plant model trajectories under control actions.

Li (1990) showed some good results using a transputer (see section 3.4.1) as a host to an A100 digital filter - a chip that in itself was a systolic architecture. The advantage of the transputer was that it was a cheap and versatile processor. The architectures in (Li 1990) were intended specifically for the calculation of controller output - the closed loop mode of operation - but a recursive least square array could also have been attached to another link of the transputer for Recursive Least Square calculations, making an all round architecture for adaptive control purposes.

The transputer was a popular choice in parallel processing for a period of time since it was cheap, had fast communications and a unique ability to communicate with other transputers while simultaneously carrying out computations on its own processor. In (Fleming 1988) the chapters are almost entirely dedicated to transputer architectures for control with applications as diverse as flight control and real-time software fault tolerance. The transputer is now obsolete in terms of current real world practical solutions, but transputer arrays are still useful for demonstrating the potential of parallel tools for the various areas in control covered in this thesis.

Another major parallel processing model is the neural network. A conventional neural network is described in detail in section 2.3. There have been a number of interesting developments as a result of research directly linked to control systems applications. These have included the combination of fuzzy logic and a neural network that makes up the architecture of the B-spline (Brown and Harris 1993) and other fuzzy networks (see for example (Chen and Teng 1995)), and neural networks that introduce transfer function models as node functions (see for example (Goulding 1991)). However, as parallel processor models, the computation at each node level is too low to make them practical for an implementation involving low cost medium- or coarse-grain processors. Neural networks in this context are viewed as models for nonlinear processes and parallelism is exploited in the control system algorithm itself. The effect of running low-grain calculations on processors of this type is discussed more fully in chapter 5.

Computational advances have not just helped to increase the speed and accuracy of existing real-world controllers, but have also helped make theoretical control schemes practical. It took thirty years before the self-tuning and adaptive control scheme built by Kalman (1958) became practical (see for example Good-

win and Sin (1984)). This is as prevalent with the advent of commodity processors for parallelisation. For example, parallel processing makes the attractive multiple model schemes of Middleton *et al.* (1988) and later Morse *et al.* (1992) practical robust adaptive controllers (schemes like these are considered in chapter 6 and chapter 8).

Other interesting applications of parallel processing to control have been in the design and simulation of control systems. Research in this area has included the attempt to generate parallel software from straight off a Simulink simulation (see for example (Baxter *et al.* 1994; Tully and Surridge 1993)). Much depends on the efficiency of the Simulink model construction as to the success of such systems.

1.2 Aims and Objectives

The following list summarises the aims and objectives of this thesis:

- To verify existing parallel schemes in a control system context.
- To identify areas of control system design or application suitable for parallelisation.
- To increase the usefullness of adaptive control by reducing/removing the limitations of such systems to the control of slow-time varying plants.
- To find and verify a general control system for the control of nonlinear plants and to parallelise such a scheme.
- To find methods through parallelisation to reduce the sampling period to an absolute minimum with a view to increasing the accuracy of identification and control of discontinuous plants (plants whose dynamics change in a discontinuous step during run time).

The rest of the thesis is structured as follows: chapter 2 describes relevant control theory; chapter 3 gives an introduction to parallel processing; chapter 4 is concerned with the verification of hardware for digital filters; chapter 5 details the parallelization of several algorithms useful in Dynamic Matrix Control; chapters 6 and 8 are concerned with the parallelization of switching and tuning schemes for linear and nonlinear plants respectively with chapter 7 detailing essential background theory underlying the methodology of chapter 8; finally, chapter 9 draws some general conclusions and gives some suggestions as to areas for further research/development.

1.3 Conclusions

This chapter has introduced the historical background to the use of computers in control with specific attention to the parallelisation of adaptive control methods. In the early days, computers were only applicable to control processes with slow process dynamics (i.e. the process industries), but increasing processor power has meant an explosion of computer usage in control. Only two existing classes of parallel application have been identified historically - systolic architectures and neural network based schemes. There is also a suggestion that parallel processing has been set aside in control due to a very fast increase in raw sequential processor speed. However, since an upper processor speed will inevitably be reached, there is still a place in the near future for relatively cheap parallel solutions (using commodity processors).

Chapter 2

Background

2.1 Introduction

This chapter provides the essential background material (with references) to the work reported in this thesis. It also gives an overview of some of the concepts and problems of constructing models of plants from available information. The thesis is mainly concerned with the construction of software models from mathematical methods of modelling and controller design described in this section. It is assumed that the control engineer is able to construct mathematical models from experimental data or existing physical models. Also, specific to each area of research in each chapter, certain assumptions will be made about the plant, such as knowledge of the order of the plant or boundedness of inputs and outputs. In chapters 6 and 8, software based schemes to extend adaptive control methods to plants that undergo large discontinuous changes during their run time are developed through the use of parallel multiple models to identify the plant. Section 2.2 describes the adaptive control viewpoint as it pertains to this thesis, where section 2.2.1 contains the mathematical preliminaries required to understand the modelling work contained

within, and section 2.2.2 gives a basic overview of Model Reference Adaptive Control. Finally, section 2.3 outlines the principles behind neural networks as used in control.

2.2 Adaptive / Self Tuning Control Schemes

Figure 2.1 shows the general structure of an adaptive self-tuning control scheme (Chisci and Zappa 1993). This is divided up into three loops:

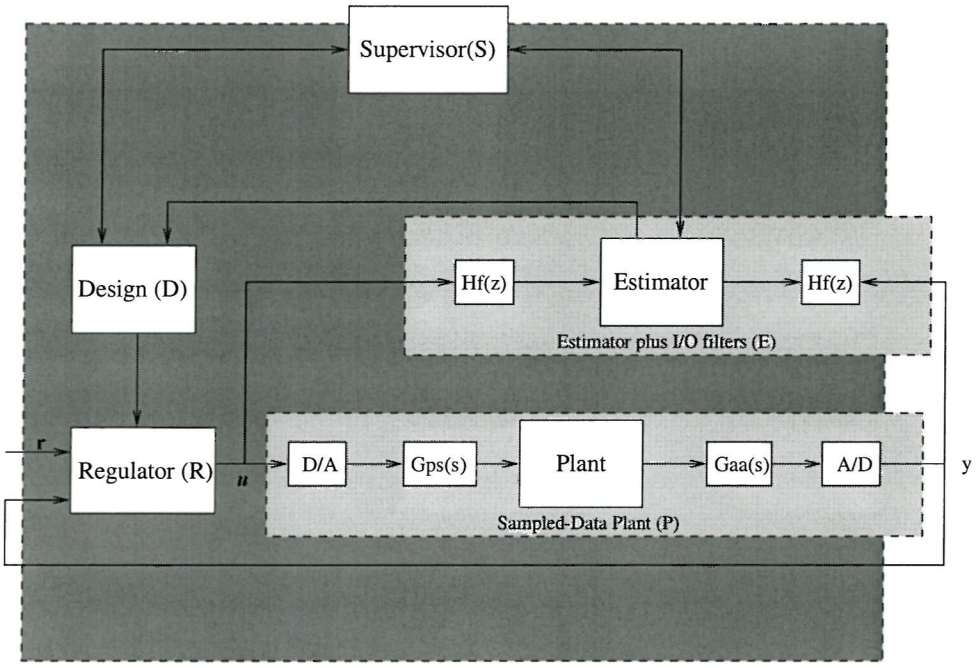


Figure 2.1: An adaptive control system

The regulation loop (R + P)

This is the standard feedback control scheme consisting of digital to analogue conversion, filtering of inputs and outputs into the plant (optional), a regulator with tunable parameters and feedback and feedforward control action.

The adaptation loop (E + D + R)

This loop consists of an on-line estimator that uses the control input (u) and the plant output (y) to tune the parameters of an adaptive model closer to those of the plant; while providing control actions by updating the regulator parameters. This loop is further described in section 2.2.2.

The supervision loop (S + D)

This can be described as the self tuning control loop. Its role is mainly to monitor the estimation of plant parameters undertaken by the estimation loop. If the convergence rate of the adaptation algorithm becomes too slow, a 'kick' can be provided (by resetting certain parameters in the adaptation algorithm). The supervision block can also provide an interface to a human operator or external devices.

2.2.1 Plant Modelling

Control of a plant depends heavily on a good plant model - the closer the model parameters are to those of the plant, the more accurate the control. In this thesis, two main types of plant model will be used. These are briefly discussed in turn below.

Differential and Discrete Linear Systems

The dynamics of a linear time invariant single-input single-output (SISO) system are described in terms of the output ($y(t)$) and input ($u(t)$) by a differential equation of the form:

$$y^n + a_{n-1}y^{n-1} + \cdots + a_0 y = b_m u^m + b_{m-1}u^{m-1} + \cdots + b_0 u \qquad (2.1)$$

where $y^n \equiv \frac{dy^n(t)}{dt^n}$, $y^0 \equiv y(t)$. Given the input $u(t)$ and the initial conditions $\{y(0), y^1(0), \cdots, y^{n-1}\}$ and $\{u(0), u^1(0), \cdots, u^{m-1}(0)\}$ the equation can (in principle) be solved for the output $y(t)$. So-called classical control systems analysis and design for continuous-time systems is undertaken in the transfer function domain using the Laplace transform and by the z-transform in the discrete time case.

Suppose, therefore, that $F(s)$ denotes the Laplace transform of the variable $f(t)$ (assumed to exist). Suppose also that (2.1) operates under zero initial conditions. Then applying the Laplace transform and rearranging gives:

$$Y(s) = G(s)U(s) \tag{2.2}$$

where the system transfer function $G(s)$ is defined by

$$G(s) = \frac{b(s)}{a(s)} = \frac{b_m s^m + \cdots + b_1 s + b_0}{s^n + \cdots + a_1 s + a_0} \tag{2.3}$$

The polynomial $b(s)$ defines the system zeros via $b(s) = 0$ and $a(s)$ defines the system poles via $a(s) = 0$ and for physical systems $n \geq m$.

Although the transfer function representation of physical systems provides an important tool for control systems analysis and design, it has certain basic limitations and, in particular, it is only applicable to linear time invariant systems. State space methods, however, do not have this restriction and are therefore more general. In essence, the state of a system is the smallest set of numbers that must be known in order that its future response to any given input can be calculated from the dynamic equation. Thus the state is a compact representation of the past history of a system which can be used to predict its future behaviour in response to any external input.

The complete solution of a differential equation of order n - such as (2.1)

- requires precisely n initial conditions, hence it follows that the state of such a system will be specified by the values of n quantities - known as the state variables.

Consider now a linear time invariant system with m outputs and l inputs - a so-called multivariable (MIMO) system. Then the state equations can be written as a set of coupled first order differential equations in the form

$$\dot{x}(t) = \frac{dx(t)}{dt} = Ax(t) + Bu(t) \tag{2.4}$$

where $x(t)$ is the $n \times 1$ state vector, $u(t)$ is the $l \times 1$ input vector, and A and B are constant matrices of dimensions $n \times n$ and $n \times l$ respectively. The output of the system is given by

$$y(t) = Cx(t) + Du(t) \tag{2.5}$$

where $y(t)$ is the $m \times 1$ output vector and C and D are constant matrices of dimensions $m \times n$ and $m \times l$ respectively. Equations (2.4) and (2.5), together with the initial state vector $x(0)$, constitute the system state space model. Setting $m = l = 1$ recovers the SISO case.

Suppose now that $x(0) = 0$. Then applying the Laplace transform to the state space model yields the system transfer function matrix (or transfer function in the SISO case) description

$$Y(s) = \left(C(sI_n - A)^{-1}B + D\right)U(s) = G(s)U(s) \tag{2.6}$$

Note also that the state vector is not unique - it is easy to see that $w = Tx$ with T nonsingular is an alternative choice for the state vector and that such a transformation leaves the transfer function (matrix) invariant. Later in this thesis properties of linear differential systems, eg controllability and observability, will

be introduced as they are needed.

As the name suggests, discrete time signals are defined only at discrete time values. Usually, the time values are equally spaced, i.e. at multiples of constant period T where $t = nT$ with n an integer. Here discrete time signals are obtained by sampling continuous-time signals or waveforms. The result of sampling $x(t)$ at $t = nT$ is denoted by $x(nT) \equiv x(n)$.

The z-transform for discrete time signals is the equivalent of the Laplace transform for continuous time signals. In the usual unilateral form it is defined (when it exists) for a sequence $\{x(n)\}$ by

$$X(z) = \sum_{n=0}^{\infty} x(n)z^{-n} = x(0) + z^{-1}x(1) + z^{-2}x(2) + \cdots \tag{2.7}$$

The mapping between the s and z domains is given by $z = e^{sT}$ and under this the stability region in the s plane, i.e. the open left-half of the complex plane, is mapped into the open unit circle in the z plane. Note that this 'full' transformation is multi-valued and hence in applications, such as the digital implementation of a continuous-time feedback control schemes, rational approximations are used, e.g. the well known bilinear transform.

It is also possible to model a discrete time system in state space form. The discrete equivalent of (2.4) and (2.5) is:

$$\begin{aligned} x(n+1) &= A_d x(n) + B_d u(n) \\ y(n) &= C_d x(n) + D_d u(n) \end{aligned} \tag{2.8}$$

Applying the z-transform now yields the z-transfer function (matrix) description:

$$Y(z) = (C(zI_n - A_d)^{-1}B_d + D_d)U(z) = G(z)U(z) \tag{2.9}$$

In chapter 4, the SISO discrete unity feedback control scheme with forward path controller will be considered, this links the error $(E(z))$ and the control input $(U(z))$ defined by

$$\frac{U(z)}{E(z)} = \frac{\sum_{i=1}^{n} a_i z^{-i}}{1 - \sum_{i=1}^{n} b_i z^{-i}} := K(z) \tag{2.10}$$

or in difference equation terms:

$$\begin{aligned} u(n+1) &= a_1 e(n) + a_2 e(n-1) + \cdots + a_n e(n+1-n) \\ &+ b_1 u(n) + b_2 u(n-1) + \cdots + b_n u(n+1-n) \end{aligned} \tag{2.11}$$

Also it follows immediately that this computation is, in effect, equivalent to that of an infinite impulse response (IIR) or ARMA filter. The only difference is that here the filter is embedded within the overall (or global) feedback loop of the scheme - a fact which has major implications for the development of parallel implementation architectures for such control schemes.

2.2.2 Model Reference Adaptive Control (MRAC) - Basics

Chapter 6 of this thesis considers the application of parallel processing tools to the implementation of multiple model based adaptive control schemes where the basic control action is applied through a model reference adaptive control scheme (MRAC). There is a very large volume of literature, both theory and applications, on this approach and in this thesis only one particular form is considered. The detail of this particular form will be given in chapter 6 and here a basic introduction to the MRAC approach is given, which mirrors closely that given in (Slotine and

Li 1991).

Generally, a *model reference adaptive control* system can be represented schematically by figure 2.2. It is composed of four parts: a *plant* containing unknown parameters, a *reference model* specifying the desired output of the control system, a feedback control law containing adjustable parameters, and an *adaptation mechanism* for updating parameters.

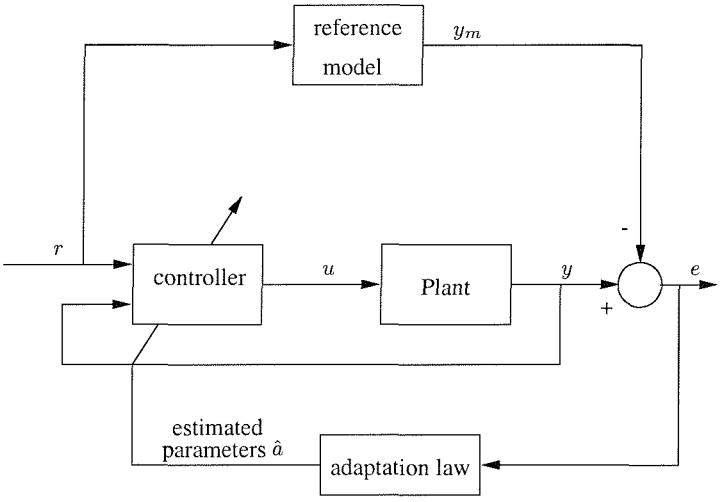


Figure 2.2: A model-reference adaptive control scheme (Slotine and Li 1991)

The *plant* is assumed to have a known structure, although the parameters are unknown. For linear plants, this means that the number of poles and the number of zeros are known, but that the location of these poles and zeros are not. For nonlinear plants, this implies that the structure of the dynamic equations is known, but that some of the parameters are not.

A *reference model* is used to specify the ideal response of the adaptive control system to an external command. Intuitively, it provides the ideal plant response which the adaptation mechanism should seek in adjusting the parameters. The choice of the reference model is part of the adaptive control system design. This choice has to satisfy two requirements. On the one hand, it should reflect the

performance specification in the control tasks, such as rise time, settling time, overshoot or frequency domain characteristics. On the other hand, this ideal behaviour should be achievable for the adaptive control system, i.e. there are some inherent constraints on the structure of the reference model (e.g. its order and relative degree) given the assumed structure of the plant model.

The *controller* is usually parameterised by a number of adjustable parameters (implying that a family of controllers may be obtained by assigning various values to the adjustable parameters). The controller should have *perfect tracking* capacity in order to allow the possibility of tracking convergence. That is, when the plant parameters are exactly known, the corresponding controller parameters should make the plant output identical to that of the reference model. When the plant parameters are not known, the adaptation mechanism will adjust the controller parameters such that perfect tracking is asymptotically achieved. If the control law is linear in terms of the adjustable parameters, it is said to be *linearly parameterised.* Existing adaptive control designs normally require linear parameterisation of the controller in order to obtain adaptation mechanisms with guaranteed stability and tracking convergence.

The *adaptation* mechanism is used to adjust the parameters in the control law. In MRAC systems, the adaptation law searches for parameters such that the response of the plant under adaptive control becomes the same as that of the reference model, i.e. the objective of the adaptation is to make the tracking error converge to zero. Clearly, the main difference from conventional control lies in the existence of this mechanism. The main issue in adaptation design is to synthesize an adaptation mechanism which will guarantee that the control system remains stable and the tracking error converges to zero as the parameters are varied. Many formalisms in nonlinear theory can be used to this end, such as

the Lyapunov theory, hyperstability theory, and passivity theory. Although the application of one formalism may be more convenient than that of another, the results are often equivalent.

2.3 Neural Networks for Control Applications - Relevant Background

Neural networks have been the subject of intense research effort over the years in many disciplines. In the general control systems area, there has been a high level of interest in the last 10-12 years. In chapter 8 of this thesis, the use of parallel processing tools to implement a nonlinear multiple model adaptive control scheme is considered. In this section, the relevant background in neural networks is given and closely mirrors that in (Levin and Narendra 1993) and the cited references.

In this thesis the term neuron will denote an operator which maps $R^n \rightarrow R$ and is explicitly described by

$$y = \Gamma(\sum_{j=1}^{n} w_j u_j + w_0) \tag{2.12}$$

where $U^T = [u_1, u_2, \cdots, u_n]$ is the input vector, $W^T = [w_1, w_2, \cdots, w_n]$ is termed the weight vector of the neuron, and w_0 is its bias. The function $\Gamma(\cdot)$ is monotone and continuous as a map $R \rightarrow (-1, 1)$ and is commonly known as the 'sigmoidal function'. A commonly used function here is $\tanh(\cdot)$.

A neural network is a set of interconnected neurons. Also if the neurons are organized in layers $l = 1, 2, \cdots, L$ and if a neuron in layer l only receives its inputs from neurons in layer $l-1$, the network is termed a feedforward neural network (see figure 2.3).

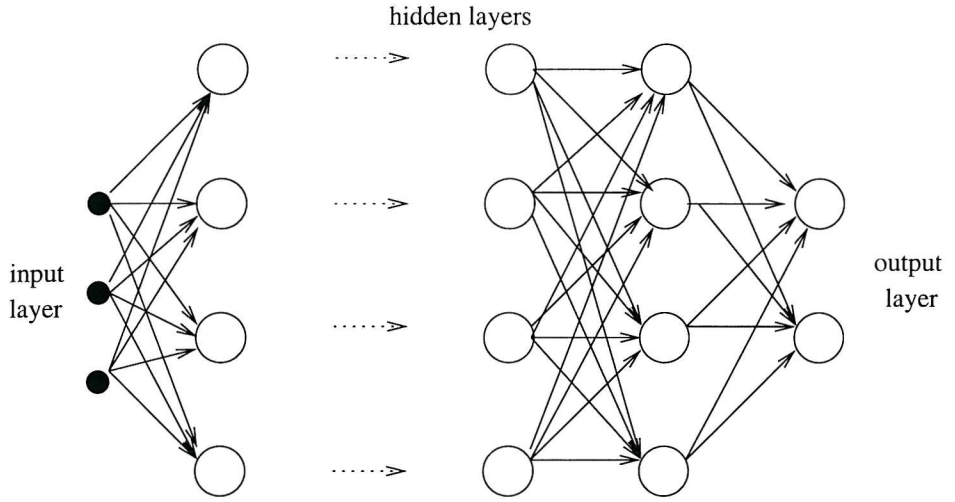


Figure 2.3: A multi-layer network (Kröse and Van der Smagt 1996)

The output of element i in layer l is given by

$$y_i^l = \Gamma\left(\sum_j w_{i,j}^l y_j^{l-1} + w_{i,0}^l\right) \tag{2.13}$$

where $[W_i^l]^T = [w_{i,0}^l, w_{i,1}^l, \cdots, w_{i,n_{i-1}}]$ is the weight vector associated with neuron i in layer l. The layer defined by $l = 0$ is commonly termed the input layer and that defined by $l = L$ the output layer. All other areas are referred to as hidden layers. The biases $w_{i,0}^l$ can be treated as additional weights associated with a neuron whose output is always equal to one.

In effect, the neural network defined here represents a specific family of parameterised maps. If there are n_0 input elements and n_L output elements, the network defines a continuous mapping $NN : R^{n_0} \rightarrow R^{n_l}$. Also to enable this map to be subjective the output layer is chosen to be linear. Such a family of networks with n_l neurons in layer l will be denoted by $NN_{n_0,n_1,\cdots,n_L}^L$. Hence, for example, if there are 2 inputs, 3 neurons in the first hidden layer, 5 in the second, and 1 output unit, the resulting network will be described by $NN_{2,3,5,1}^3$.

Multilayer feedforward neural networks are universal approximators which

make them very powerful tools in function approximation - it has been shown (Cybenko 1989; Hornik *et al.* 1989) that any continuous mapping over a compact domain can be approximated to arbitrary accuracy by a feedforward neural network with one hidden layer. Hence given any $\epsilon > 0$, a neural network with a sufficiently large number of nodes can be found such that

$$||f(x) - NN(x)|| < \epsilon, \ \forall x \in D \tag{2.14}$$

where f denotes the function to be approximated, and D is a compact domain of a finite dimensional normed vector space with norm denoted by $||\cdot||$.

Suppose now that $u \in D$ is a given input. Then the network approximation error for this input is given by

$$e(u) = f(u) - NN(u) \tag{2.15}$$

and training $NN(\cdot)$ to closely approximate f over D is equivalent to minimizing

$$I = \int_D ||e(u)||\, d\, u \tag{2.16}$$

The training procedure for the network is implemented as follows where $\theta \in W$ denotes a generic parameter (or weight) of the network.

1. The network is presented with a sequence of training data in the form of input-output pairs.

2. Following each training example, the weights of the network are adjusted according to the rule

$$\theta(k+1) = \theta(k) - \eta(k)\frac{\partial I}{\partial \theta}|_{\theta=\theta(k)} \tag{2.17}$$

Stochastic approximation theory (Ljung 1977) now guarantees that, if the step size $\eta(k)$ satisfies various conditions (e.g. that it a sufficiently small step - unique to each system - and if it is variable that it does not rise over time), I will converge to a local minimum with probability 1. If the performance surface is unimodal then this fact implies that the global minimum is achieved.

The so-called back propagation algorithm (see section 2.3.1) provides a recursive method to calculate these gradients in recursive networks, i.e. the partial derivatives with respect to the weights in layer $l-1$ can be calculated recursively given the ones of layer l (Narendra and Parthasarathy 1991; Rumelhart *et al.* 1986). Also weight adjustments can be performed at each time step or in a batch mode. In the latter case, the error function depends on the errors due to a finite set of input vectors.

Introducing feedback connections into the network makes it recurrent and in this case the behaviour cannot be described as a static mapping from the input to output spaces. Instead, the output will exhibit complex temporal behavior that depends on both the current states of the neurons and the input. To avoid algebraic loops (and make such a feedforward connection physically meaningful), a delay (i.e. z^{-1}) must be added. Hence from the systems theory standpoint, the states of the system (in its state space model representation) consist only of those neurons which have a delay at their output.

A natural performance criterion for the recurrent network is

$$I(k) = \sum_k ||y(k) - \hat{y}(k)||^2 \equiv \sum_k ||e(k)||^2 \tag{2.18}$$

i.e. the sum of the squares of the errors between the plant output vector ($y(k)$) and the network ($\hat{y}(k)$). Also a training algorithm, termed dynamic back propagation,

has been developed to train a recurrent network to follow a temporal sequence. This algorithm is based on the fact that the dependence of the output of a dynamical system on a parameter is itself described by a recursive equation. The latter, in turn, contains terms that depend both explicitly and implicitly on the parameter. For a detailed treatment of this algorithm see, for example, (Narendra and Parthasarathy 1991).

2.3.1 Error Back Propagation

At the output layer the error function is calculated as:

$$I = \frac{1}{2}\sum_{k}(d_k - y_k^{(L)})^2 \tag{2.19}$$

where d_k is the desired output at output node k and $y_k^{(L)}$ is the output at the k^{th} node of the output layer, i.e.

$$y_k^{(L)} = \Gamma\left(\sum_{j} w_{jk}^{(L-1)} y_j^{(L-1)}\right) \tag{2.20}$$

Here, $\Gamma(\cdot)$ is taken as the sigmoid function, i.e.

$$\Gamma(x) = \frac{1}{1+e^{-x}} \tag{2.21}$$

The object of back propagation is to calculate Δw_{jk}. At the output level (L), assuming a constant training rate η, this is:

$$\Delta w_{jk}^{(L-1)} = \eta \frac{\delta I}{\delta y_k^{(L)}} \frac{\delta y_k^{(L)}}{\delta w_{jk}^{(L-1)}} \tag{2.22}$$

or

$$\Delta w_{jk}^{(L-1)} = \eta(d_k - y_k^{(L)})y_k^{(L)}(1 - y_k^{(L)})y_j^{(L-1)} \tag{2.23}$$

or

$$\Delta w_{jk}^{(L-1)} = \eta e_k^{(L)} y_j^{(L-1)} \tag{2.24}$$

with

$$e_k^{(L)} = (d_k - y_k^{(L)})y_k^{(L)}(1 - y_k^{(L)}) \tag{2.25}$$

For the lower (hidden) layers it is necessary to calculate:

$$\Delta w_{ij}^{(l-1)} = \eta \frac{\delta I}{\delta w_{ij}^{(l-1)}} = \eta \frac{\delta I}{\delta y_j^{(l)}} \frac{\delta y_j^{(l)}}{\delta w_{ij}^{(l-1)}} \tag{2.26}$$

and at the last hidden layer ($l = n - 1$) this is:

$$\Delta w_{ij}^{(L-2)} = \eta \frac{\delta I}{\delta w_{ij}^{(L-2)}} = \eta \frac{\delta I}{\delta y_j^{(L-1)}} \frac{\delta y_j^{(L-1)}}{\delta w_{ij}^{(L-2)}} \tag{2.27}$$

and $\frac{\delta I}{\delta y_j^{(L-1)}}$ is obtained from:

$$\begin{aligned} \frac{\delta I}{\delta y_j^{(L-1)}} &= \frac{1}{2}\sum_k (d_k - y_k^{(L)})^2 \\ &= -\sum_k \left((d_k - y_k^{(L)} \times \left(\Gamma' \left(\sum_j w_{ij}^{(L-2)} y_i^{(L-2)} \right) w_{jk}^{(L-1)} \right) \right) \end{aligned} \tag{2.28}$$

or

$$\frac{\delta I}{\delta y_j^{(L-1)}} = -\sum_k \left((d_k - y_k^{(L)}) \times \left(\Gamma'(y_k^{(L)}) w_{jk}^{(L-1)} \right) \right) \tag{2.29}$$

$$\frac{\delta I}{\delta y_j^{(L-1)}} = -\sum_k \left((d_k - y_k^{(L)})(y_k^{(L)}(1 - y_k^{(L)}))w_{jk}^{(L-1)} \right) \tag{2.30}$$

$\Delta w_{ij}^{(L-2)}$ then becomes:

$$\Delta w_{ij}^{(L-2)} = \eta \left[\sum_k e_k^{(L)} w_{jk}^{(L-1)} \right] \times y_j^{(L-1)}(1 - y_j^{(L-1)}) y_i^{(L-2)} \tag{2.31}$$

or

$$\Delta w_{ij}^{(L-2)} = \eta e_j^{(L-1)} y_i^{(L-2)} \tag{2.32}$$

where

$$e_j^{(L-1)} = \left(\sum_k e_k^{(L)} w_{jk}^{(L-1)} \right) \times y_j^{(L-1)}(1 - y_j^{(L-1)}) \tag{2.33}$$

Due to the recursive nature of the differentiation involved, this weight change for lower layers is applicable for all layers $(L-1) \cdots 0$. (Simply replace L with l in equations (2.32)-(2.33)).

2.4 Conclusions

This chapter has introduced some essential theory used throughout the thesis. It has described the general idea of adaptive and self-tuning control, distinguishing the regulation, adaptation and supervision loops. Various plant modelling techniques were then introduced developing the underlying differential equations into transfer function and state-space representations. The general concept of Model

Reference Adaptive Control was also introduced. Finally the theory of neural networks was described in a control system environment.

Chapter 3

Parallel Processing

3.1 Introduction

Parallel processing is the concept of applying more processors to a task with the aim of computing that task more quickly. Parallel computers may be classified into four architecture types based on the combination of whether they work on multiple or single data streams; and whether they perform multiple or single instructions on the data. This is summarised in table 3.1.

The SISD computer is the original sequential Von Neumann architecture which includes few of the currently used computers - with most sequential computers containing parallel components as part of their normal operation (e.g. instruction pipelining). The architecture is shown in figure 3.1. The basic Von Neumann architecture had no cache, which was introduced to alleviate the Von Neumann

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 3.1: Parallel computer architecture classification (Trew and Wilson 1991)

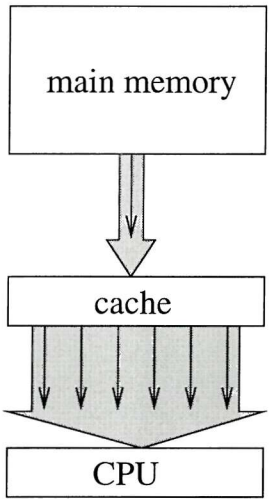


Figure 3.1: The SISD architecture (Dowd 1993)

bottleneck - a limit on speed related to processing time and memory access time. The cache is made up of a small amount of faster (more expensive) memory which data is loaded into if not in the cache already. Programs can be optimised by making maximum usage of data in the cache. It should be noted that the Central Processing Unit (CPU) is a redundant concept in parallel processing and is renamed the Processing Element (PE).

The MISD computer is not thought to be of much practical use and no computer of this type has been built. It basically runs multiple programs on the same datum. The SIMD architectures are formed from a large number of processors which all perform the same instruction on different data. These might be hardwired computers that are specifically designed for one purpose - for example the systolic architectures discussed in chapter 4, or fine grain architectures that perform small logical operations. The extremes of this type are the Distributed Array Processor (DAP) (Hunt 1989) - made up of a few thousand 1 bit processors linked by a single control unit (the MCU onto which the program is loaded) - and the Connection Machine made up of many thousands of DAP-like processing

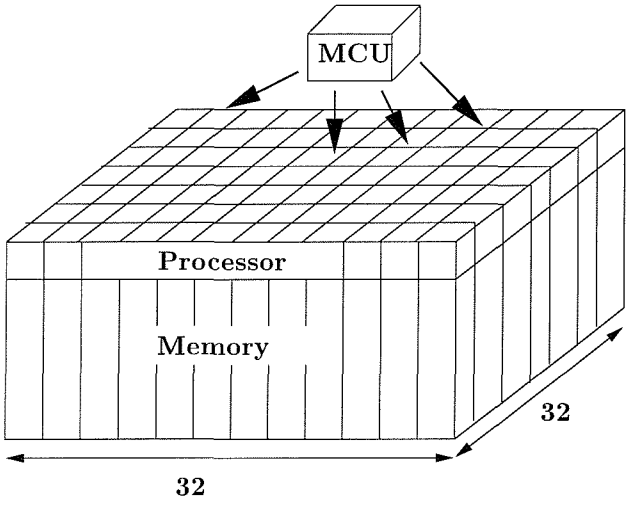


Figure 3.2: The Distributed Array Processor (DAP) - an SIMD architecture (Hunt 1989)

elements with a random routing component that helps to eliminate communication bottlenecks (Trew and Wilson 1991). The basic DAP architecture is shown in figure 3.2 with each processor allocated either 32kbits or 64kbits of memory each - a total of 4Mb or 8Mb for the whole array. SIMD architectures can be extremely efficient when applied to applications such as image processing where the same calculation is performed on each pixel of an image; but become less efficient in applications where computational load is not uniformly distributed (such as resolutions of shadow and reflection in ray-tracing algorithms).

The MIMD architectures are a course/medium grain approach. A relatively small number of processors (compared to SIMD architectures) run individual programs on different data streams. In practice, these programs can be the same for each processor, but with no requirement of synchronisation or master control outside of communications between processors. As each processor can hold whole programs, it is possible to recycle large quantities of software that are already in existence - provided the compiler for the software concerned exists on the machine. This is the great advantage of the Cray T3D computers that link at least 32 pow-

erful vector computers together (scalable to 2048 vector processors) and implicitly parallelise loops in Fortran code. Another option is to produce cheap processors that can be linked together via fast communication buses and thereby produce faster cost-effective machines (e.g. linked NT PC clusters). The individual processor architectures are of the type shown in figure 3.1 with the CPU replaced with a PE and some form of communication link that allows message routing to other processors.

An implicitly parallel machine automatically parallelises code as part of the compiler process. An explicitly parallel machine provides parallel coding components as part of the computing language (e.g. Occam - the native language to the transputer - and the parallel C libraries provided with Parix (Parsytec 1993)).

The architecture types can be further classified as either shared memory or distributed memory systems. Shared memory systems (see figure 3.3) allow all processors to read and write to the same memory blocks. This can cause allocation problems since it is not possible to stop processors changing the values of data from one processor to another in mid-calculation. This is solved by the introduction of *mutual exclusivity* where processors still share the same memory but are not allowed to share variables (or specific bytes in memory). This essentially sub divides memory amongst processors on demand. Distributed memory systems provide individual memory on each processor. This allows whole programs to be run on each processor with no further latency problems caused by checking memory access requests as in shared memory machines (the conventional Von Neumann bottleneck normally associated with single processors still exists).

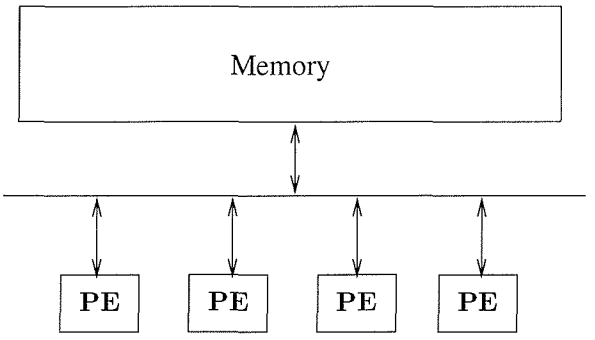


Figure 3.3: Shared memory computers

3.2 A Use for Parallel Processing

This thesis is concerned with the development of cost effective real-time solutions to various areas of control. Faster and faster sequential machines are being developed all the time, but in a world in which high performance computing is talked about in terms of gigaflops[1], the technology to achieve this on a sequential machine can be too expensive. This points to solutions involving a number of cheap processors running in parallel. Massively parallel computers are ignored here. Instead attention is focused on the development of tools with many uses (users) which implies a relatively (compared to hugely expensive massively parallel machines) cheap set of solutions in many cases of practical interest.

Parallel processing is specifically useful in control engineering because the time between sampling periods is greatly limited. Between consecutive samplings, plant models have to be updated and control inputs have to be calculated. A parallel solution can reduce the computation time, allowing more complex plant models to be included in the scheme at no extra cost in time; or, if need be, to reduce the time between sampling periods, i.e. to increase the sampling rate. The key is real-time solutions. An algorithm may work in simulation, but if computational overheads are too high to make the solution practical in the real world it is useless.

[1] FLOPS - Floating Point OPerationS

An extremely important point to note at this stage is that not all algorithms are suitable for parallelisation. Parallel algorithms that are suitable for some massively parallel computers are too fine grain (requiring a very small amount of computation per processor) to be suitable for commodity MIMD processors. An example of this is the DMC work in chapter 5. The communication overheads to distribute the required data across the processor topology is extremely high and as a result speed-up was only achieved after communication was minimised and the topology was optimised. However, in this case if a calculation required double precision data, the communication message lengths are nearly double and speed-up is no longer possible. An algorithm that is inherently sequential may still benefit from parallelisation if some or all of the sequential steps are suitable for parallelisation. The general rule in ascertaining whether an algorithm is likely to be parallelisable on medium- or coarse-grain processors is to look at the ratio between communication overheads and computation per processor. If this ratio is too high speed-up will be low or, if communication is too high, the process will take longer in parallel than on a single processor. These concepts are discussed in the next section.

3.3 Scalability

The basic concept of scalability is: How many processors can be applied to a problem before no further speed-up can be achieved? This depends on the parallelism of the problem. If an algorithm is easily decomposable into a number of subtasks, which require little global data to begin work, then the scalability of an algorithm is likely to be good. Scalability simply describes how close to an n-times speed-up an algorithm is when more processors are added to a solution. Normally there

will be a point at which speed-up begins to tail off, as individual subtasks become smaller, and communication overheads, as a result, become more significant.

Numerically, the perfect speed-up can be expressed as:

$$T_N = \frac{T_1}{N} \tag{3.1}$$

where T_1 is the time taken to complete an algorithm on a single processor and T_N is the time taken to complete the same algorithm on N processors. There are several reasons why T_N can be much more than T_1/N:

(i) *Shared resources.* If the network of processors is controlled by a single master processor, then a bottleneck will occur while message data is constructed and routed through the network. Competition for external (input/output) devices can also slow times down significantly (it is standard to return data to a single processor and record it there).

(ii) *Communication time.* This is the time taken to communicate the required data to processors before computation can start (including time taken to synchronise processors, if needed). It is important to note that the transputer is unusual in its ability to overlap communications and computations, most processor types will have to receive all data before computation can start.

(iii) *Message Latency.* This is the time taken to set up messages before they can be sent. This can be quite substantial. For example, an MPI linux cluster (connected by a fast ethernet connection capable of transmitting $100 Mbits/sec$) has a latency of about $150\mu s$. With the maximum bandwidth of about $11 Mbytes/sec$, this amounts to a minimum message size (to minimise delays caused by latency) of $1 Kbyte$ which is greater than 100 double precision numbers (at 8 bytes each) (Takeda *et al.* 1999).

(iv) *Sequential Components.* Parts of the algorithm that must be performed sequentially limit the overall speed-up. The maximum possible speed-up can be expressed numerically as:

$$S_N = \frac{T_s + T_p}{T_s + \left(\frac{T_p}{N}\right)} \tag{3.2}$$

where T_s is the total time taken to compute the parts of the algorithm that have to be performed sequentially, and T_p is the total time taken to compute the parts of the algorithm that can be performed in parallel.

Scalability can now be thought of as an efficiency term:

$$Scalability = \frac{T_1}{NT_N} (\leq S_N) \tag{3.3}$$

Scalability should always, where possible, be calculated using the original sequential code before it has been parallelised. The sequential code should also be in an optimised state since an optimised sequential algorithm can run much faster than an equivalent sequential algorithm in an unoptimised state.

3.4 The Parallel Platform

The software simulations in this thesis (i.e. all work except chapter 4) have been implemented on an array of 16 T800 transputers (see section 3.4.1). It should be noted that these processors are slow by current standards but allow an easy comparison between parallel and sequential results. The array is divided into boards of 2 processors each, with each board connected to a back plane, forming a network that is fully reconfigurable, allowing the modelling of any topology[2],

[2] A topology is a graph (or tree) of processors, in which the usual aim is to minimise the relative distance between processors and/or avoid communication 'bottlenecks'.

within the bounds of the 16 nodes. Timing studies have shown that communication between processors on the same board takes less time than communication between processors on different boards. As a result, section 5.3 includes an investigation into optimising topologies by minimising across-board communication. The simulations in chapter 8 were run on a PowerXplorer array (which is also fully reconfigurable) with direct comparisons made with the transputer array in the performance analysis in section 8.6.

All code is compiled within the Parix Environment (which is described in section 3.4.3). As far as the programmer is concerned, Parix is a parallel form of Unix which provides standard compilers (C and Fortran) but with additional libraries to support communication between processors, 'virtual' topology construction, and positional identification of processors within a network.

3.4.1 Transputers

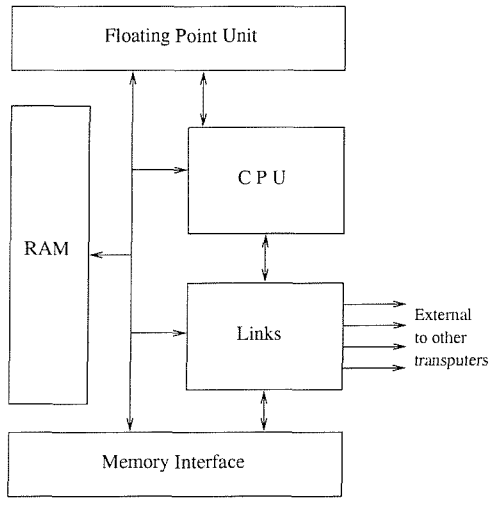


Figure 3.4: The T800 Transputer

The transputer (figure 3.4) is a RISC[3] implementation built around the CSP

[3]Reduced Instruction Set Computers

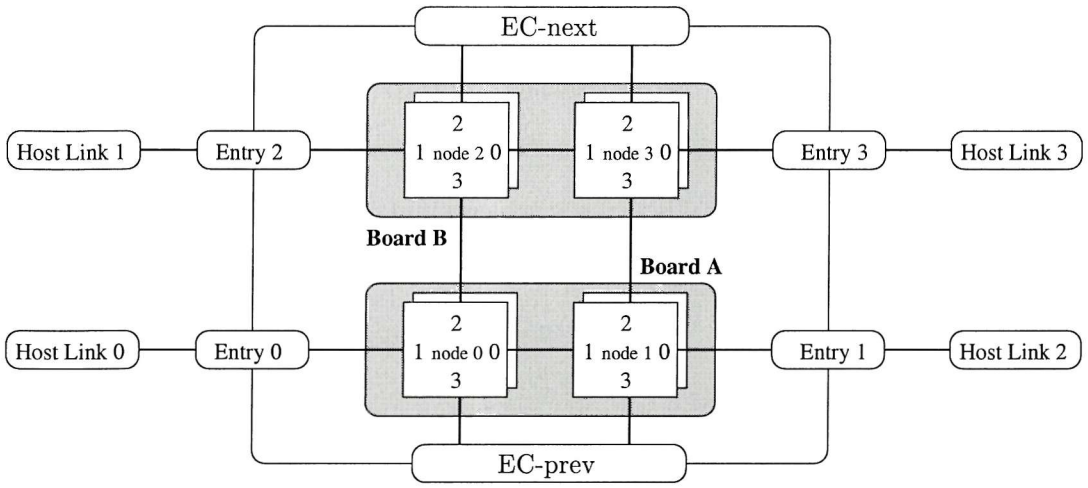


Figure 3.5: A 2 × 2 partition on the PowerXplorer array (Parsytec 1994)

model concept (Hoare 1978). The native language of the transputer is Occam which, like CSP, is constructed from the fundamentals of assignment ($a = 1$ etc), and input and output communication. However, since this research is concerned with the development of generic tools, software portability is more important, and programming in a version of C (Parix) lends itself more conveniently to this. The transputer (at the machine code level) still aims to complete the three most common operatives (*load*, *store* and *add constant* combinations of which account for 60 - 80% of instructions) in one clock cycle (May and Shepherd 1990). Transputers can also communicate with one another (via four bi-directional communication links) while simultaneously carrying out computations.

3.4.2 The PowerXplorer Architecture

The basic PowerXplorer array component is shown in figure 3.5. In similar manner to the transputer array, processors appear two to a board and are connected to other processors via four bi-directional communication links. The communication in the system is provided by T805 transputers (a close relative of the T800) while

processing power is provided by an MPC 601 chip. From the results provided in section 8.6 and the 1×1 partitions in tables 8.3 and 8.8 it can be seen that the MPC 601 is approximately 15 times faster than the T800. Larger arrays can be built from these components which are connected via the EC-next, EC-prev and the entry points (Parsytec 1994).

3.4.3 The Parix Implementation

Parix and the T800 Transputer (Parsytec 1993)

The Parix implementation (Parsytec 1993) uses a software router (when dealing with T800's) to emulate a T9000/C104 network when interpreting the virtual links, which aims to produce an optimal routing between processors. The T9000 was a proposed upgrade of the T800, and the C104 is a routing chip that can be connected to 32 transputer links. As far as the programmer is concerned, this means that the same functionalities exist for application purposes. The only difference depends on how the software router interprets communications (which are already slower on the T800 than on the T9000).

The Parix Programming Model (Parsytec 1993)

Static qualities:

(i) The main program is duplicated on all processors.

(ii) Processors can be identified by a number giving their individual position in a network.

(iii) A library of pre-programmed topologies is available.

Dynamic qualities:

(i) Allows the creation of 'virtual links' between arbitrary processes and processors.

(ii) Supports the object-oriented concept (C++) of threads (a communication between objects)

(iii) Supports synchronous and asynchronous communication.

(iv) User defined virtual topologies can be created

Links to external devices:

(i) External devices can be accessed via a host processor

Also:

(i) No network description language is required. Topologies are created using graph modelling.

3.5 Conclusions

This chapter has introduced those concepts common to parallelisation: data and instruction parallelisation through the description of SIMD and MIMD computer models; explicit and implicit parallelisation; and shared and distributed memory machines. Where possible, example machines of each type were described. The use of parallelisation was then discussed in a control theory context before the concept of scalability was introduced mathematically and discussed. Finally, the machines specifically used for the implementations of the methods in this thesis are described together with a brief outline of the Parix operating system model.

Chapter 4

Architectures for Real-Time Feedback Controllers

4.1 Introduction

The work in this chapter is primarily concerned with verifying systolic/wavefront architectures described in (Li 1990). Systolic architectures are defined as follows ((Kung 1988) and (Li 1990)):

A systolic architecture is a network of computer processors exhibiting (a) synchronisation of data flow with computation (the processors are controlled by a global clock); (b) modularity and regularity (there are a limited number of cell types which appear often); (c) spatial and temporal locality; (d) the speed-up increases at a linear rate. For example, N cells will give a near N-times speed-up - the only losses are due to the communication of data to the next cell in the array (see figure 4.1).

The wavefront architecture differs from the systolic array only in that the

control of dataflow is self-timed and data-driven, rather than by a global clock. This eliminates the need for temporal locality as data operations are triggered by the arrival of data from neighbouring elements.

The main advantage of systolic/wavefront architectures is that multiple computations are permitted for each memory access. This is particularly advantageous for problems where multiple operations are carried out on the same data inputs. A good deal of control problems belong to this category. The general idea is shown in figure 4.1.

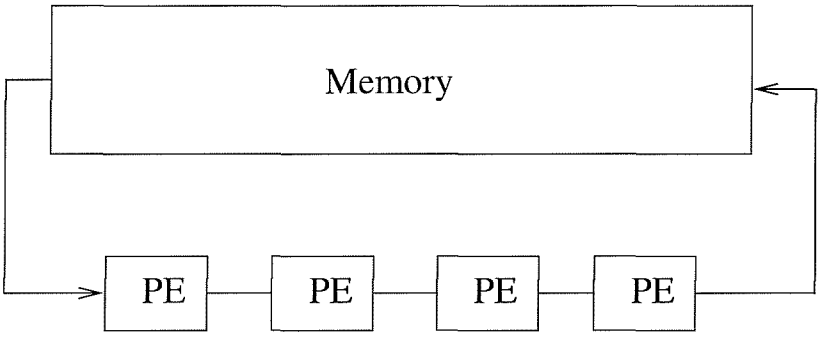


Figure 4.1: Systolic architectures (Kung 1988)

The benefits of these systolic/wavefront architectures, when compared with other special purpose architectures are as follows (Kung 1988):

(i) high-level parallelism and pipelining.

(ii) multiple use of data which need only be accessed once per sample.

(iii) high speed operation.

(iv) simple timing circuitry.

(v) an ability to map high-level algorithms onto VLSI architectures.

(vi) cost effectiveness

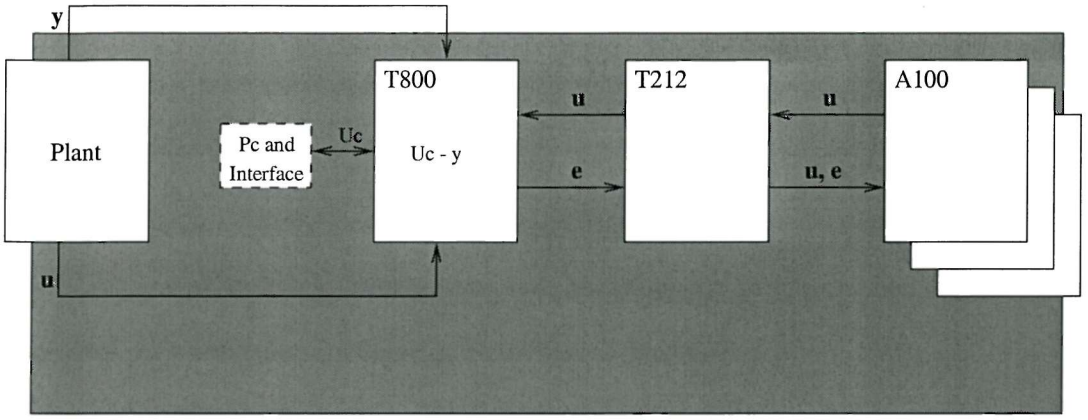


Figure 4.2: T800 and A100 Systolic/Wavefront architecture

These architectures have been applied to many areas of research including, of most interest to this work, Digital Signal Processing and Recursive Least Squares ((Kwan 1987), (Gaston *et al.* 1994)). In effect, the controller in the digitally implemented feedback control scheme considered here can be viewed as a Digital Signal Processing filter embedded in a global feedback loop. Also, Recursive Least Squares is a classic approach in system identification (see section 6.2).

4.2 An Heterogeneous Architecture for Digital Feedback Control

Several architectures are developed in (Li 1990) but ultimately, through Li's own research, the architectures evolved into that shown in figure 4.2. This architecture involves transputers supervising the operation of a DSP chip (the A100 chip).

For the purpose of verification, attention is focused on one relatively simple case; that of a discrete unity negative feedback control scheme where the forward

path controller is described by:

$$U(z) = \frac{\sum_{i=1}^{n} a_i z^{-i}}{1 - \sum_{j=1}^{m} b_j z^{-j}} = H(z) \tag{4.1}$$

which in difference equation terms is:

$$\begin{aligned} u(k+1) = a_1 e(k) + a_2 e(k-1) + \ldots + a_n e(k-n+1) + \\ b_1 u(k) + b_2 u(k-1) + \ldots + b_m u(k-m+1) \end{aligned} \tag{4.2}$$

where

$$e(k) = r(k) - y(k) \tag{4.3}$$

and $r(k)$ is the reference signal that the output $y(k)$ is required to follow. The A100 DSP chip has a traversal filter structure. In other words it is designed to execute the calculations defined by a finite impulse response (FIR) filter. Equation (4.2) is an infinite impulse response filter, which can be regarded as being constructed from two sections of an FIR filter.

An individual A100 chip is capable of computing a difference equation involving up to 32 coefficients (i.e. $i = 16$) or, perhaps more importantly, of computing several subtasks in parallel. Each processing cycle is triggered by an external 'GO' signal which allows the operation of the A100 to be program controlled - in the architecture in figure 4.2, the signal is to be generated by the host transputer. The A100 also provides two coefficient registers (the current coefficient register (CCR) and the update coefficient register (UCR)) which allow the host transputer to update the filter coefficients every 'GO' cycle. This is important in an application such as adaptive/self-tuning control where controller coefficients are changing continuously and need to be updated on-line. The architecture of the A100 is shown

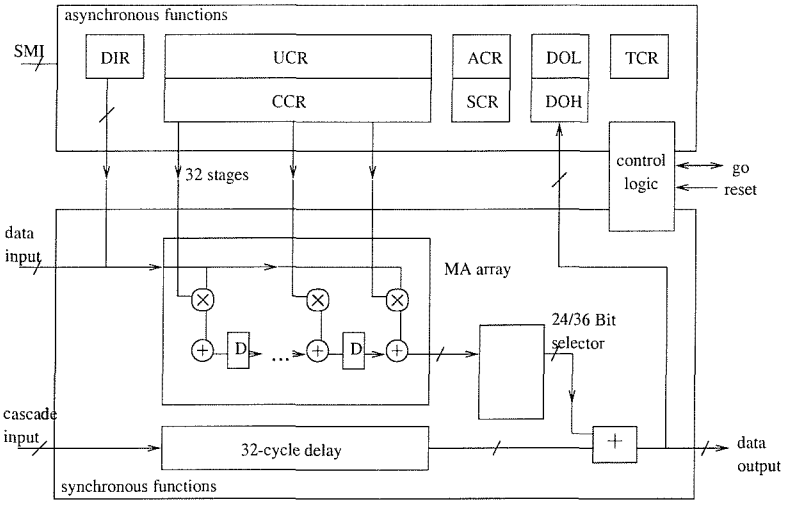


Figure 4.3: Block diagram of the IMS A100

in figure 4.3.

The A100 expects all coefficients to be multiplied together with the incoming data (in control terms it is more specifically designed for an FIR filter) so it is necessary to calculate one half of the difference equation first (i.e. the b's triggered by the arrival of u). This means interleaving the coefficients in the A100 registers as shown in figure 4.4, with the a's awaiting the arrival of the error data (e) in the UCR. When the other half of the difference equation calculation is triggered by the arrival of e, the coefficients are swapped. They are swapped every cycle, triggered by the arrival of new data, and in order to ensure the correct timing, and a correct transfer function, the coefficients are interleaved with zeros. An alternate solution is to use two A100 chips in the architecture, with one chip calculating each half of the difference equation.

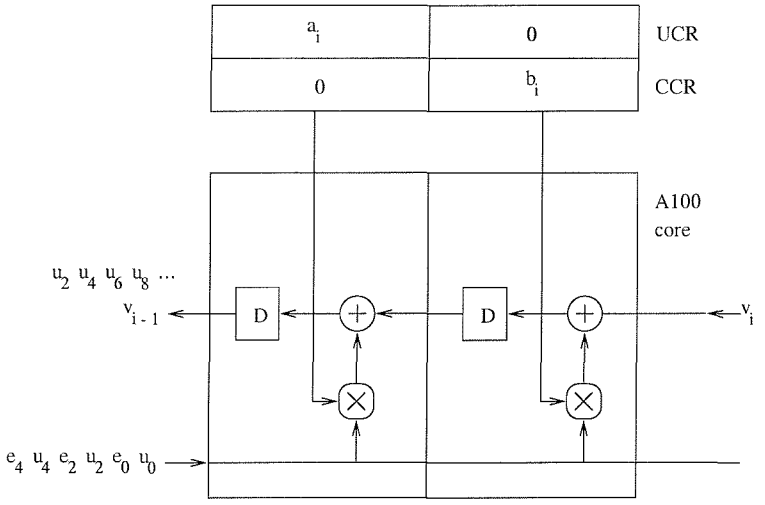


Figure 4.4: Type 1 cell using two stages of A100 processing core

4.3 Validation

4.3.1 Transfer Function Validation

This verification was demonstrated in previous work, and appears in (Li and Rogers 1993) and (Brown *et al.* 1995), but is duplicated here for convenience.

It is necessary, before any further investigation, to check that the new architecture (figure 4.2) has the transfer function H(z). One method of verification, used by (Lin 1986) among others, is the snap shot method to check the behaviour of the implementation either after every cycle, or after several cycles. This can prove difficult to use in practice and a new method was developed in (Li 1990). The basics of which can be summarised as follows:

(a) By inspection, write the difference equations that describe the signal flow properties of the cell.

(b) Each cell in the array is identical, hence iterate these equations n-times, where n is the number of cells.

(c) The transfer function is correct if this iteration converges to the difference equation describing the input/output behaviour of the controller.

(d) To check the expandability of the array, repeat (c) with n replaced by $n+1$.

This is, however, intended for a fine grain approach (i.e. where all the cells are the same). For the purpose of this verification, first consider a single cell in the homogeneous architecture shown in figure 4.4. The time domain behaviour is described by the difference equation

$$v_{i-1}(k+1) = v_i(k) + a_i e(k) + b_i u(k) \tag{4.4}$$

Iterating n-times produces:

$$\begin{aligned} v_o(k+1) = v_n(k-n) + a_1 e(k) + a_2 e(k-1) + \ldots + a_n e(k-n+1) + \\ + b_1 u(k) + b_2 e(k-1) + \ldots + b_n u(k-n+1) \end{aligned} \tag{4.5}$$

which is the same as equation (4.2) since $v_0 = u$ and $v_n = 0$. Expandability is also easily proved.

It is also possible to apply these techniques to figure 4.2. However, since it is inherently time varying, the verification must be applied to two complete 'GO' cycles.

The data corresponding to $e(k)$ are delayed by one cycle before input to the array. Hence, in transfer function terms, a single cell or filter section is governed by $E^1 = z^{-1}E$, where E^1 denotes the state of the error delayed by one GO cycle.

This yields the following equation, when applied to the cell architecture:

$$\begin{bmatrix} E^1 \\ V_{i-1} \\ U \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ a_i z^{-1} & z^{-2} & b_i z^{-2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} E^1 \\ V_i \\ U \end{bmatrix} \tag{4.6}$$

Expanding this to n-cells, for a complete description of the architecture gives: (using $E^1 = z^{-1}E$)

$$\begin{bmatrix} E^1 \\ U \\ U \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \sum_{i=1}^{n} a_i \zeta^{-i} & \zeta^{-n} & \sum_{i=1}^{n} b_i \zeta^{-i} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} E^1 \\ V_i \\ U \end{bmatrix} \tag{4.7}$$

which can be rewritten as:

$$U = \sum_{i=1}^{n} a_i \zeta^{-i} E + \sum_{i=1}^{n} b_i \zeta^{-i} U \tag{4.8}$$

where $\zeta^{-1} = z^{-2}$. Providing the plant output is sampled and the controller output is collected at every even cycle, this equation corresponds to equation (4.2). The time domain analysis yields:

$$v_{i-1}(2k') = v_i(2k' - 2) + a_i e(2k' - 2) + b_i u(2k' - 2) \tag{4.9}$$

which corresponds to equation (4.2) provided $2k' = k$, where k' is the new discrete time scale for the interleaved data streams.

4.3.2 Tractability

A problem is described as being tractable if its run time is a polynomial function of the input size. A problem is intractable if its run time is no less than exponentially

related to the input size. This is an important consideration because if a problem is intractable it will take as long to perform a calculation on a sequential computer regardless of increases in processor speed. In short, intractability is a property of the problem and its solution and not the computing model - therefore, there is no point in parallelising it. If the relationship between the run time and the input size is represented by P(n), this can be reduced down to the question:

Does $P(n) = a_0 + a_1 n + a_2 n_2 + \ldots + a_k n_k$?

The input size in this case is the number of coefficients + the number of inputs of u and e, i.e.

Input size: $2k + k + k = 4k$.

If the process time for 16-bit real multiplication is m (as the process time for 32-bit addition is the same as 16-bit multiplication on the A100, this is also m) and the time for assignment then the process time is:

$P(n) = 2m.2k + a$

i.e. *Process time* = 2m. *Input size* + a

which is a polynomial of order 1. Therefore the problem is tractable, and, in theory, speed-up is possible on the architecture. It should be noted that this equation is valid only because each data item need only be accessed once from memory. Since one new value of u and e appears each sampling period, the equation complexity (which would be intractable if all previous values of u and e had to be accessed every sampling period) is greatly reduced.

4.3.3 Timing Considerations

The purpose of this part of the investigation was to determine the answer to two questions:

(a) What is the minimum possible length of a 'GO' cycle?

(b) How much data can be extracted in the sampling period (limited by the length of the 'GO' cycle)?

Three assumptions have been made in order to answer these:

(a) The A100 is always going to communicate its results after the completion of the calculation, which allows any communication between the A100 and the transputer to be declared dead-lock free.

(b) The plant is assumed to be always available for sampling.

(c) As long as two parties are ready for communication, the actual transmission time between them is assumed to be negligible.

The minimum period of the 'GO' cycle can be expressed as:

$$T_w = 2(T_m + T_a + T_b) \tag{4.10}$$

where T_m and T_a are the word level multiplication and addition times respectively and T_b is the transmission time (including program overheads).

The A100 takes the same time to calculate a 16-bit multiplication as a 32-bit addition at 80 Million FLoating point Operations a second (MFLOPS). This means for a controller of order $n < 16$, the processing time of the A100 is therefore $(2n-1)/(8\times10^7)$ (sec) and on an even cycle the transputer used requires $n(T_1+T_2+$

T_3+T_4) (sec), where T_1 and T_2 are the times for input and output communications with the plant respectively, T_3 is the time required for assignment, and T_4 is the time required to access the reference signal. In Occam, the native language of the transputer, T_1, T_2 and T_3 are all fundamentals and require one clock cycle to perform since the plant is always open to communication.

On an odd 'GO' cycle the minimum length of the 'GO' cycle is $n(T_s+T_a)+B_c$, where T_s is the time required for integer addition / subtraction, and B_c denotes the time before communications can take place after the previous step. Under the given assumptions, B_c is negligible. Which leaves the minimum length of the 'GO' cycle, and hence the sampling period, as

$$max((n(T_s + T_a)); (2n - 1)/(8 \times 10^7)) \tag{4.11}$$

Unless the calculation speeds of the T800 transputer and the A100 chip vary considerably, the significant term will arise from the A100 term. This time is an absolute minimum time, based on manufacturing guide lines, and hence the actual length of the 'GO' cycle will depend on efficiency of coding.

4.4 Conclusions

This chapter has been concerned with the development of verification techniques for the parallel architectures introduced in Li (1990). These architectures are of use in key elements of generic feedback control schemes based on adaptive / self-tuning controllers. If these parallel architectures prove viable it will be possible to use them in the computations required within critical length sampling periods.

Quicker calculation of plant models means shorter sampling periods (and hence more plant information can be gathered) and / or more computationally complex plant models can be constructed.

The timing cycles have been modelled and theoretical minimum length sampling periods have been presented.

The parallel architectures have proved to be highly parallel and scalable. The software controls of the process are a potential pitfall and success depends on the efficient coding of the problem - paying particular attention to communication overheads - a common problem in parallel processing.

Any control system that controls a plant that can be modeled by an nth order difference equation could benefit from the architectures in Li (1990).

Chapter 5

Parallelisation of a Dynamic Matrix Controller

5.1 Introduction

This chapter continues the investigation into the development of cost effective solutions for the parallelisation of control schemes. Where the previous chapter investigated systolic architectures whose cost effectiveness came from their simplicity; this section looks at software implementations for situations where a limited number of general purpose processors is available.

A desire to create plant models of increased size and complexity has begun to push sequential machines to their limits. The ultimate performance limitation in control is often the length of the sampling period and practically this needs to be reduced to a minimum, to provide the most accurate control possible. This ideal, coupled with the greatly increased computation required for more complex models, can easily push the computation overheads beyond the capabilities of sequential machines.

Dynamic Matrix Control can suffer from these problems, particularly when attempting control of multiple input multiple output (MIMO) systems where modelling requires the computation of many linear differential equations (in the form of discrete/difference equations as a result of sampling). One result of this has been limited 'real world' applications in the multiple-input multiple-output (MIMO) case.

Cost effective solutions (with only a limited number of processors available) in this area have largely been ignored. The software implementation investigated uses a simple dynamic matrix control algorithm which, in processing terms, reduces to matrix vector multiplication.

DMC has been used extensively in the chemical industry, for example in the control of high-purity distillation columns (Chien 1996). Chien uses empirical data to develop a simple 1st order nonlinear model of a distillation column, where nonlinear functions depend on operating systems (where processing measurements are taken) and upper and lower temperature bounds. There has been a good deal of research into robust design of Model Predictive Control algorithms (MPC of which DMC is a member) which, because most chemical processes are nonlinear, leads to the solving of complex programming problems. (Sarimveis *et al.* 1996) is one example of such work. Genceli and Nikalaou (1995) found that in order to provide rigorously designed MPCs for nonlinear processes with ensured stability and performance robustness, a high model accuracy is required. Finally, (Lee *et al.* 1994) was able to extend the conventional step response model, to systems with white measurement noise - via state estimation techniques - without increasing algorithm complexity.

5.2 Dynamic Matrix Control (DMC)

This section gives an overview of the basic ideas behind Dynamic Matrix Control when applied to MIMO systems.

Basic DMC involves the prediction of the response of each system output to inputs, constructed by feedback control, over the 'prediction horizon' using discrete convolution models. These predicted responses can than be compared with the actual responses once the sampling period has elapsed, and used to calculate a vector of prediction error across the prediction horizon (N). Here N denotes the number of convolution models, i.e. outputs.

These prediction errors (if available) can then be used to compute control inputs such that the next predicted response approaches the desired set point trajectory. Suppose that at instant k, $U(k) \in R$ denotes the vector of control inputs, $Y(k) \in R_m$ the vector of outputs, $R(k) \in R_m$ the vector of reference trajectories, the subscript i indicates a particular channel in the corresponding vector; and

$$e = [e_1(k+1), \ldots, e_1(k+N), e_2(k+1), \ldots, e_2(k+N), \ldots, e_m(k+1), \ldots, e_m(k+N)] \tag{5.1}$$

the vector of predicted errors over the prediction horizon N. Then the control input vector for the next sampling period is computed as:

$$U(k+l) = Ke \tag{5.2}$$

where $(p = lN)$

$$K = \begin{bmatrix} k_{11} & \cdots & k_{1p} \\ \vdots & & \vdots \\ k_{m1} & \cdots & k_{mp} \end{bmatrix} \tag{5.3}$$

is the $m \times p$ matrix of feedback control gains. The computation of e is based on fast control actions and does not include current and future control inputs. The elements of the vector e are computed thus:

$$e_j(k+l) = e_j(k) - W_{ij},\ l \leq j \leq N,\ l \leq i \leq m \tag{5.4}$$

where, in effect, W_{ij}, includes future predictions of Y_i based on previously implemented control signals and

$$E(k) = R(k) - Y(k) \tag{5.5}$$

The formula for calculating W_{ij} is

$$W_{ij} = \sum_{b=1}^{j} X_{ib} \tag{5.6}$$

where

$$X_{ib} = \sum_{d=1}^{l} \sum_{c=b-1}^{T} H_{id}(c) u(d, k+l-c) \tag{5.7}$$

With $H_{id}(c)$ denoting the ith coefficient of the open loop impulse response of the ith output channel to the dth input channel, and $u(d, k+1-c)$ is the sample of the dth input channel at sampling instant $k+1-c$. The open loop impulse can be obtained from a step response test on the (assumed open loop stable)

process to be controlled. Here the values of the unit step response are denoted by $Y_i(0), Y_i(1), \ldots Y_i(T)$ with a sampling period of Δt and it is assumed that $Y_i(k) = 0, k \leq 0$. The integer T is termed the model horizon and the settling time of the process is taken to be $T\Delta t$. Finally,

$$\begin{gathered} h_{id}(c) = Y_i(c) - Y_i(c-1), 1 \leq c \leq T \\ h_{id}(0) = 0 \end{gathered} \tag{5.8}$$

Computation time of this scheme can be seen to be most dependent on the calculation of all X_{ib} quantities. This computation reduces to a simple matrix vector multiplication:

$$R = S.M \tag{5.9}$$

where the matrix S is of dimension $mN \times lt$ whose elements are all the coefficients of the impulse responses of the outputs to each input, also the vector M has the structure:

$$M = [u_1(i-1), \ldots, u_1(i+1-T), \ldots, u_t(i-1), \ldots, u_t(i+1-T)]^T \tag{5.10}$$

5.3 Implementation and analysis

(This section is adapted from (Brown *et al.* 2000))

To produce a model of parallel matrix-vector multiplication, it is necessary to assume a model architecture. Since the machine used for the test calculations is transputer based, the model is a transputer-like model, but the primary results can be applied to other parallel architectures as, in the basic model, features which are not available on most parallel architectures (e.g. the ability to overlap

communications and calculations on a single processor) are not employed. In particular, the following assumptions are made.

1. The processor array can be configured so that any two processors can communicate directly.
2. The speed of communication is the same between all directly connected processors.
3. There are no overlaps of communication on a single processor - i.e. if there are more than one (set of) link(s) on a processor, only one can be in operation at a time.
4. There are no overlaps between communications and computation.
5. Data packages between processors are of sufficient length so that latency effects are negligible.

These are of course a set of idealised conditions, but allow the construction of a simple 'best case' model which is not tied specifically to any particular architecture, and so should have general applicability. Note that here the reference to a link is to a physical link; and that a block of data will be specifically tracked through a processor array, without the assumption of a routing harness - in which much of the cost and timings would not be under direct control.

In addition it is assumed that:

6. If there are p 'worker' processors then the problem can be broken into p subproblems of equal size.

This last assumption, is not strictly necessary, but it ensures load-balancing, and simplifies the analysis that follows.

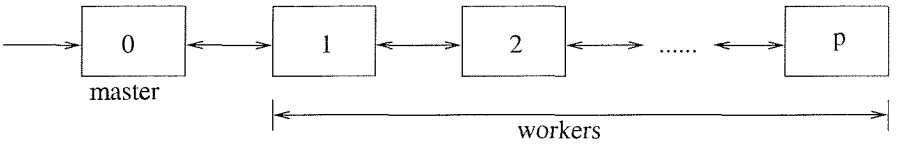


Figure 5.1: Model architecture: Pipe

In terms of the machine architecture, it is assumed there is a master processor (processor 0) which assembles the matrix and vector - and p worker processors (processors 1 to p) - which perform all calculations. Further it is assumed that all elements of the matrix and vector are assembled by the master processor before any data is transferred to the workers, and that there is only one link between the master and the workers (the effects of relaxing these assumptions will be discussed below). The last assumption has the effect that, in terms of the behaviour of the configurations of the worker processors, the architecture is equivalent to a simple pipe (figure 5.1).

With a single worker, the total time taken to perform the calculation, T, can be split into the time taken for communications, T_m (which includes the time taken to return the results to the master), and the time for the computations, T_c, with $T = T_m + T_c$. Suppose now that there are p (> 1) workers. Breaking the problem up into p subproblems of equal size (assumption 6), the first step in the algorithm is to transmit the first block of data from processor (proc) 0 to proc 1; the second step is to transmit this data from proc 1 to proc 2; and, on the third, to transmit it from proc 2 to proc 3, while simultaneously transmitting the second block of data from proc 0 to proc 1, etc.

In tabular form:

$$\begin{aligned}
step\ 1: &\quad 0 \xrightarrow{D_1} 1 \\
step\ 2: &\quad 1 \xrightarrow{D_1} 2 \\
step\ 3: &\quad 0 \xrightarrow{D_2} 1,2 \xrightarrow{D_1} 3 \\
step\ 4: &\quad 1 \xrightarrow{D_2} 2,3 \xrightarrow{D_1} 4 \\
\vdots &\quad \vdots \\
step\ 2p-1: &\quad 0 \xrightarrow{D_p} 1,\ldots\ldots,p-1 \xrightarrow{D_1} p
\end{aligned}$$

where D_i denotes the *ith* block of data and the arrow a transfer of data between processors. Computations are then performed on all workers simultaneously, followed by the transmission of the results back to the master by a procedure which is essentially the reverse of that given above. Ignoring any overheads in the communications from sending p 'small' sets of data rather than a single 'large' set (assumption 5) the total elapsed time for the calculation is now:

$$\begin{aligned}
T_l &= (2p-1)\frac{T_m}{p} + \frac{T_c}{p} \\
&= \left(2 - \frac{1}{p}\right) T_m + \frac{1}{p} T_c
\end{aligned} \tag{5.11}$$

With this model, there will be a gain from using multiple workers if $T_l < T_m + T_c$, which reduces to:

$$T_m < T_c \tag{5.12}$$

i.e. the time for communications should be less than for the computation when there is a single worker. Perhaps more realistically, if $T_l < T_c$ is required, i.e. that the cost of performing the calculations on the master processor is greater than

distributing the calculations, the following equation is obtained:

$$\frac{T_m}{T_c} < \frac{p-1}{2p-1} \tag{5.13}$$

assuming that the master processor is of the same speed as the workers, and that it does not perform any of the calculations. Trivially, equation (5.13) gives $T_m < 0$ for p = 1, but in practice T_m/T_c varies from 1/3 when $p = 2$ to 1/2 as p tends to infinity. In all cases it is significantly more restrictive than equation (5.12), and for a range of values of T_m/T_c there is a minimum number of processors required to achieve speed-up from distributing the calculation.

The scenario presented above, in which one of the processors controls the procedure with the others performing the calculations, is common in distributed processing. However, here it is possible to make more efficient use of the processor array by including the master processor in an equal share of the computation. In which case the total elapsed time becomes:

$$(2p-1)\frac{T_m}{p+1} + \frac{T_c}{p+1} \tag{5.14}$$

Requiring this to be less than T_c produces:

$$\frac{T_m}{T_c} < \frac{p}{2p-1} \tag{5.15}$$

which again varies from 1/3 ($p = 1$) to 1/2 ($p \to \infty$).

The second scenario with $T_l < T_c$ may seem more realistic, but the first is also of interest as it models the situation in which the data is gathered externally to a processor array before being passed along for the calculations, particularly in a situation in which the data can be assembled at least as fast as it can be processed. The case in which processor 0 has more than one link will be considered below.

A key feature in the model presented above is that processor 1 is always active, from which it follows that any configuration of the worker processors will produce the same (model) processing time as a simple pipe. However, in practice, if the worker processors have more than one link it would be sensible to configure the processor array as a mesh, for example, and send successive blocks of data down different links as there will be natural idle periods in any route used which should make the algorithm more tolerant to communication bottlenecks.

From the above, it is clear that in theory significant gains from using multiple processors will not be achieved unless the computation time T_c is significantly greater than the communication time T_m. The results are likely to be worse than those predicted. The estimate for the computation time of each processor will be reasonably accurate. However, the communications time is a 'best case' minimum estimate which ignores the effects of overheads and latency.

Now these predictions are to be compared with test calculations. The machine used for all calculations is a Parsytec Multicluster II with 16 25Mhz T800 transputers, each with 4Mb memory. The operating system of the array is Parix, Parsytec's parallel version of Unix, and is coded in C, with communications implemented using routines supplied with Parix. For reference, during the largest runs, the data transfer rate along a single link was 8.0 Mbits / sec, and the computations were performed at 0.38 Mflops in single precision (32-bit) and 0.33 Mflops in double precision (64-bit). These figures are in line with the machine specifications.

To proceed further, the decomposition of the algorithm onto the processor array must be specified. The obvious ways of doing this are to split the matrix S by rows or by columns. Here the latter is chosen, and each row is split so that each processor performs part of each inner product for every row, before returning the values to processor 1 - which completes the summation, and finally returns

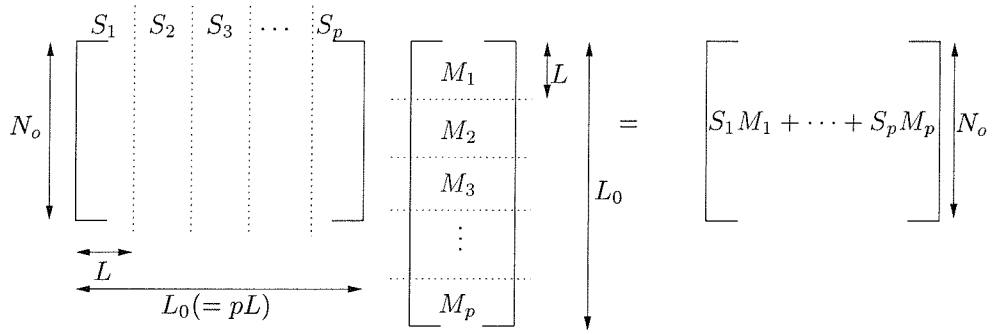


Figure 5.2: Division of the matrix vector calculation

the result to the master processor. If the matrix S is of dimension $N_0 \times L_0$, and the vector M is $L_0 \times 1$, ($L_0 = pL$ is assumed where L is an integer) then S is decomposed thus:

$$S = [S_1, S_2, \ldots, S_p] \tag{5.16}$$

where each of the S_i is an $N_o \times L$ matrix, with M similarly decomposed into $pL \times 1$ vectors, $M_1, \ldots, M_p$. Processor i will calculate the $N_o \times 1$ vector S_iM_i, and processor 1 will sum these vectors to obtain SM. The division of the matrix vector calculation is shown in figure 5.2.

When estimating the total elapsed time for the calculation for the parallel algorithm, it has been assumed, in effect, that the number of arithmetic operations and the total data transfer through the processor array is the same, regardless of the number of workers. This is not strictly true. For the algorithm described above, with a single worker, proc 1 receives $N_oL_o+L_o$ values and returns N_o values. Thus, if α is the average time for transmission of a single value, equation (5.17) is obtained.

$$T_m = \alpha(L_oN_o + L_o + N_o) \tag{5.17}$$

However, if $p > 1$, then processor 1, which controls the procedure, will receive $L_o N_o + L_o$ values from processor 0, forward $(p-1)L(N_o+1)$ values to the other workers, receive $(p-1)N_o$ in return and finally return pN_o values to processor 0. Hence, the total time for communications can be estimated as

$$\left(2 - \frac{1}{p}\right) T_m + \alpha \left(p - 2 + \frac{1}{p}\right) N_o \tag{5.18}$$

rather than $(2 - 1/p)T_m$ as used above. However, the relative increase given by equation (5.18) is $\Theta((p-1)/L_o)$, which, for the type of problem of interest here, will be small. There is a similar small increase in the computational work performed by proc 1 (an extra $(p-1)N_o$ additions).

There are a number of reasons why a simple model is used (as represented by equations (5.11) (5.15), and (5.17)) to analyze the numerical results rather than the exact model as given by equation (5.18). Firstly, the errors will be small since problems in which L_o and N_o are large and p relatively small are primarily of interest. Secondly, the simple model can be used for any sensible parallel decomposition of the model, whereas the deviation from it will depend on the details of the parallel algorithm. Thirdly, the errors should be much less than the variation which can arise from differences in the configuration of the hardware used for the calculation.

The Parsytec machine is reconfigurable, and any two processors can be linked together, but the time taken for communications between any two processors will depend on the routing. In particular, the machine has eight boards, each with two processors, and there can be a significant difference in the time taken to transmit the same amount of data between two processors on the same board and those on different boards. For example, a test calculation was performed using single precision arithmetic for a square matrix with $N_o = 576$, a single worker, and

using processors 0 and 1 in the array, which are on the same board, the complete calculation took 3.2 seconds, while using processors 0 and 6, the calculation took 4.2 seconds. This was the worst case encountered in this calculation, all other combinations tried gave better agreement. In the results given below for the pipe, no attempt has been made to optimise the configuration to minimise the effects on the algorithm of the different rates of data transfer. Communication in the test calculation and all examples presented here is *packeted* and sent a row of the matrix at a time.

N_0	Run time (secs)	% Time on computation
18	0.0058	35.0
36	0.0203	39.1
72	0.0760	41.4
144	0.2972	42.1
288	1.1762	42.4
576	4.6650	42.7
1152	18.5828	42.9

Table 5.1: Square matrix $N_o \times N_o$, $p = 1$, double precision.

N_0	Run time (secs)	% Time on computation
18	0.0104	4.9
36	0.0274	7.6
72	0.0981	8.2
144	0.3675	8.6
288	1.4234	8.8
576	5.6903	8.8
1152	22.7529	8.8

Table 5.2: Square matrix $N_o \times N_o$, $p = 4$, double precision.

N_0	Run time (secs)	% Time on computation
18	0.0205	1.4
36	0.0445	2.2
72	0.1145	3.2
144	0.4005	3.6
288	1.5672	3.6
576	6.2199	3.6
1152	24.7817	3.6

Table 5.3: Square matrix $N_o \times N_o$, $p = 9$, double precision.

Tables 5.1 - 5.3 give representative times for runs with a master processor

and one, four and nine workers respectively, using double precision arithmetic and the original scenario in which the master processor does not perform any of the computations. The results show that for small values of N_o, as expected, start-up and overheads are significant, but as N_o increases, the proportion of total time spent on the numerical computations asymptotes to a constant value with $T_c/(T_m + T_c) \cong 0.43$. Since $T_m > T_c$, the analysis above predicts that total time for the calculation will increase if more workers are used. Using the values given in table 5.1 , equation (5.11) predicts that the total time for $N_o = 1152$ will be 20.5 seconds with $p = 4$ and 20.9 seconds if $p = 9$. The values given in tables 5.2 and 5.3 for these cases are larger than the predicted values (22.8 seconds and 24.8 seconds respectively), not surprisingly since the model is a best case analysis, but clearly the trend is as predicted.

There is no point in considering this case further, as the results demonstrate unequivocally that, with the processor array employed, the quickest way to perform the calculation for any size of matrix is to use a single processor, and not try to distribute the calculation.

Better results would be expected for the single precision case, where the time spent on arithmetic computations will decrease by roughly 10% while, for sufficiently large problems, the communications time should halve. Tables 5.4 - 5.6 are similar to 5.1 - 5.3, except that single precision arithmetic has been used. For a single worker, the proportion of the time spent on the computations now converges to approximately 0.58 seconds, and hence the model predicts a marginal decrease in the total time taken as p increases. From table 5.5, there is a small decrease in the total time when $p = 4$ for $N_o \geq 288$, but there is a small increase when $p = 9$ (table 5.6), which is almost certainly associated with delays in the communications.

N_0	Run time (secs)	% Time on computation
18	0.0039	47.8
36	0.0143	50.4
72	0.0525	54.3
144	0.2012	56.4
288	0.7948	57.0
576	3.1590	57.3
1152	12.5694	57.6

Table 5.4: Square matrix $N_o \times N_o$, $p = 1$, single precision.

N_0	Run time (secs)	% Time on computation
18	0.0087	5.4
36	0.0205	9.3
72	0.0562	13.0
144	0.2043	14.0
288	0.7775	14.7
576	3.0328	14.9
1152	12.1601	14.9

Table 5.5: Square matrix $N_o \times N_o$, $p = 4$, single precision.

N_0	Run time (secs)	% Time on computation
18	0.0192	1.4
36	0.0392	2.3
72	0.0877	3.9
144	0.2306	5.6
288	0.8164	6.3
576	3.2090	6.3
1152	12.7738	6.3

Table 5.6: Square matrix $N_o \times N_o$, $p = 9$, single precision.

Even with single precision, there is a drastic increase in the total time taken as p increases when N_o is small. This highlights the detrimental effects of the overheads in the communications with small problems, and the need to transmit the data in sufficiently large packets so as to minimise the effects of these overheads, in accordance with assumption 5. A simple test in which each value in the matrix and vector was transmitted separately further illustrated this point. With $p = 1$, and $N_o = 576$, the total time for the calculation was now 68.1 secs for single precision, and 69.8 secs for double precision, as compared with 3.2 and 4.7, respectively, as given in tables 5.4 and 5.1, when each message between processors 0 and 1 consisted of a complete row of the matrix (or the vector).

It is clear that with the present model, essentially a simple pipe, where communications and computations cannot be overlapped, with the hardware and software employed for the calculations, the fastest way to perform the matrix-vector multiplication is to use a single processor. However, if the restrictions placed on the model are relaxed to take advantage of some of the features of transputers, then a more efficient parallel algorithm can be derived. In particular, by dropping the assumption of a master processor with a single link to workers, the processors can be configured into a two-dimensional mesh in such a way that the total communication time for the parallel algorithm can be almost halved. For convenience, assume that the processors have been configured into a square mesh (e.g. 2×2, 3×3) with the top left processor designated as proc 0, which sends the data to the other processors, assembles the final results, and performs an equal share of the matrix-vector multiplication. Then, providing successive blocks of data are sent down different links, e.g. the first through the 'east' link the second through the 'south' link, the third 'east' etc (see figure 5.3), the total communications time will reduce to $((p-1)/p)T_m$, while the computation time can be estimated

as T_c/p, where p is the total number of processors in the mesh. As before, there will be errors in these estimates due to decomposition of the algorithm, but again, these will be insignificant if the problem is sufficiently large. A routing strategy must be derived to distribute data for any particular mesh and this can be done in a number of ways. The only general rule to be applied is the obvious one that the nodes on the mesh furthest from proc 0 should be populated first. A sample routing strategy for a 3×3 mesh is shown in figure 5.3. With this model, there will be a gain if

$$\left(1 - \frac{1}{p}\right) T_m + \frac{1}{p} T_c < T_c$$

i.e. if

$$T_m < T_c \tag{5.19}$$

For double precision $T_m > T_c$, and hence it will, once again, be less efficient to parallelise the algorithm. However, for single precision $T_c < T_p$ and some modest gains might be achieved from parallelisation. Tables 5.7 - 5.10 show results for a single processor, and 2×2, 3×3, and 4×4 grids, respectively. There are a large number of entries in table 5.7, to provide a reasonable number of cases for comparison on the other grids. For small values of N_0, the time spent on the computation is less than 100% because of overheads in initiating the calculation, but this time soon becomes insignificant. As expected, for small values of N_o the total time increases with the number of processors.

Assuming $T_c/(T_m + T_c) = 0.58$ (see Table 5.4), the model predicts that there will be gains of 21%, 25% and 26% with $p = 4,\ 9,\ 16$, respectively, when compared with the computation time on a single processor. For $p = 4$, there is a slight

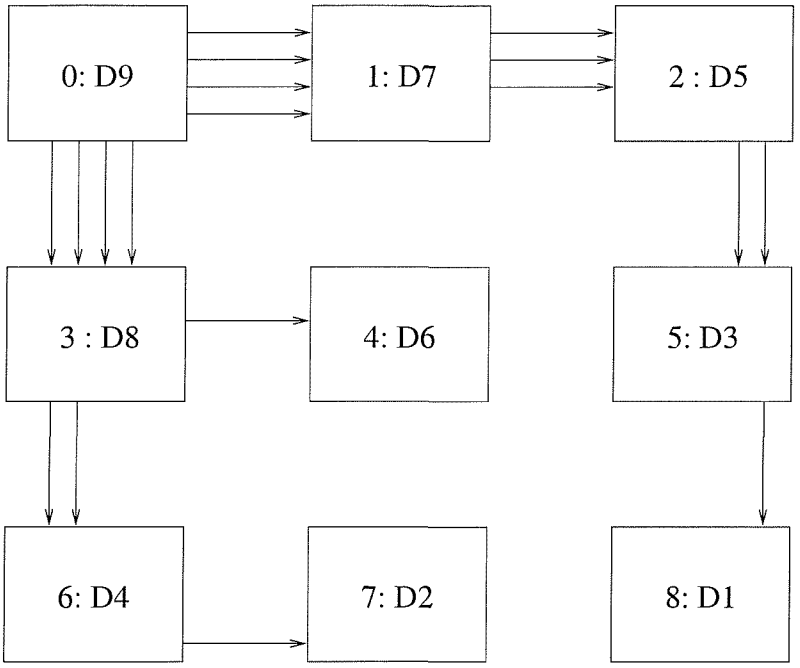


Figure 5.3: Model architecture: Mesh. Inside each box is the processor number and the block of data processed on each node, e.g. processor 4 processes the sixth block sent by processor 0.

N_0	Run time (secs)	% Time on computation
16	0.0015	96.9
18	0.0018	97.5
32	0.0056	99.2
36	0.0070	99.4
64	0.0219	99.8
72	0.0277	99.8
128	0.0871	100.0
144	0.1102	100.0
256	0.3476	100.0
288	0.4398	100.0
512	1.3887	100.0
576	1.7574	100.0
1024	5.5520	100.0
1152	7.0265	100.0
4096	88.7974	100.0
4608	112.3823	100.0

Table 5.7: Mesh, Square matrix $N_o \times N_o$, $p = 1$, single precision.

N_0	Run time (secs)	% Time on computation
16	0.0044	9.1
32	0.0103	14.1
64	0.0297	19.0
128	0.1087	20.3
256	0.3885	22.51
512	1.4667	23.74
1024	5.6948	24.41
4096	89.8204	24.72

Table 5.8: Mesh, Square matrix $N_o \times N_o$, $p = 4$, single precision.

N_0	Run time (secs)	% Time on computation
18	0.0132	1.9
36	0.0270	3.2
72	0.0648	5.0
144	0.1801	7.0
288	0.7638	6.5
576	2.6351	7.5
1152	9.7188	8.1
4608	148.1125	8.4

Table 5.9: Mesh, Square matrix $N_o \times N_o$, $p = 9$, single precision.

N_0	Run time (secs)	% Time on computation
16	0.0146	1.0
32	0.0286	1.5
64	0.0607	2.5
128	0.1466	4.0
256	0.4138	5.4
512	1.4439	6.1
1024	5.0232	7.0
4096	71.6002	7.8

Table 5.10: Mesh, Square matrix $N_o \times N_o$, $p = 16$, single precision.

increase in total time for the two largest values of N_o, 1024 and 4096, but a 10% and 19% decrease, respectively, with $p = 16$. However, with $p = 9$, there is a significant increase for all values of N_o (greater than 30% for the largest values used). Thus, while the results for $p = 16$ are broadly in line with expectations (some gain but not as much as predicted) those with $p = 4$ and $p = 9$ are not. The obvious explanation for this is delays in transmission of data due to a suboptimal configuration of the mesh. As shown above, there can be significant differences in the time taken to transmit a block of data between different pairs of processors in the array, and a test was performed comparing processors 0 and 1 with processors 0 and 3 for the master-single worker scenario with $N_o = 576$. The former took 3.2 sec and the latter 4.2 sec, close to the worst case found (0 and 6). Since the link from 0 and 3 is one of the two most active when $p = 9$, (sending four blocks of data, the same as the link from 0 to 1) this delay will clearly cause a significant degradation of the algorithm as a whole.

The relatively good agreement between the predictions and the results for $p = 16$ suggest that this mesh is a good, if not optimal, configuration for the machine, and that better results might be obtained for $p = 4$ and $p = 9$ by using a submesh of the 4×4 mesh used with $p = 16$. To be precise, for $p = 16$, the processors were labeled from 0 to 15 and processors $0-3$, $4-7$, $8-11$ and $12-15$ were used for rows $1-4$ in turn, providing a natural mapping between the mesh and the processor array and using all processors in the array. In generating tables 5.8, and 5.9, processors $0-3$ and $0-8$ were used, respectively, with the first 2/3 for the first row of the mesh etc. Instead, the top left part of the 4×4 mesh could be used for the smaller mesh, i.e. processors 0, 1 (row 1) and 4, 5 (row 2) for $p = 4$, and $0-2$ (row 1), $4-6$ (row 2) and $8-10$ (row 3) for $p = 9$. The results of these calculations are given in tables 5.11 and 5.12, which show much

better agreement with predictions. For $p = 4$ (table 5.11), there is a 16% gain with $N_o = 1024$ and 17% with $N_o = 4096$, as compared with the predicted 21% gain. For $p = 9$, the gains are 15% for $N_o = 1152$ and 19% for $N_o = 4608$, against 25% from the theory. In general, given the simplicity of the model, the agreement between the predictions of the model and the test results is good.

From tables 5.10-5.12 it is possible to deduce a minimum desirable message length. With $N_o = 1024$ there is a gain in speed (as compared to the single processor version, table 5.7) of 16% for $p = 4$ and 10% for $p = 16$. In comparison, with $N_o = 4096$, the gains were 17% for $p = 4$ and 19% for $p = 16$, which indicates that by this stage the communications are reasonably efficient. With $N_o = 1024$ and $p = 4$, each packet of data is 1024 bytes long, but only 256 bytes long for $p = 16$. With $N_o = 4096$ the packets are 4096 and 1024 bytes for $p = 4$ and $p = 16$ respectively. Also for $p = 9$ and $N_o = 4608$, the gain is 19% and the packet length 2048 bytes. From the results, the conclusion is that a data packet needs to be at least 1Kbyte long for efficient communications using the Parix message passing routines. The largest problem considered had $N_o = 4608$, which would require approximately 85Mbytes memory to store the matrix on processor 0, far more than the 4Mbytes available. However, this case was easily simulated by overlapping rows of the matrix - the numerical results are of course meaningless, but the timings produced are valid. In a similar manner to that for the mesh, the configuration for the pipe could clearly be improved to produce better agreement with the theory. This has not been done as the results presented above show clearly that, even in the best case, there would be little gain when using a pipe.

So far, although a transputer array has been used for test calculations, generic processor arrays have been modelled, ignoring specific features available on few other (if any) general purpose microprocessors. Now the effect of using all the

N_0	Run time (secs)	% Time on computation
16	0.0038	10.4
32	0.0092	15.8
64	0.0260	21.5
128	0.0918	24.0
256	0.3230	27.1
512	1.2077	28.8
1024	4.6659	29.8
4096	73.7713	30.3

Table 5.11: Submesh, Square matrix $N_o \times N_o$, $p = 4$, single precision.

N_0	Run time (secs)	% Time on computation
18	0.0092	2.8
36	0.0188	4.7
72	0.0466	7.0
144	0.1318	9.6
288	0.4616	10.7
576	1.6080	12.2
1152	5.9677	13.1
4608	90.7798	13.8

Table 5.12: Submesh, Square matrix $N_o \times N_o$, $p = 9$, single precision.

facilities of the transputer are considered. First, it has been shown that there are significant differences in the data transfer rates between different pairs of processors in the machine used in the calculations reported above. These variations could be eliminated by using a hard-wired processor array with direct communications between the processors. Also, by coding in Occam, it would be possible to get close to the maximum speed of 20Mbits/sec on each link, rather than the 7.7Mbits/sec achieved using Parix. With Occam, there would be a similar increase in the computation rate, not to the nominal speed of 2Mflops, but at least to 1Mflop with the type of calculations performed in this study. Since the expected proportional increase is roughly the same for both communications and computations, the arguments above on the merits or otherwise of parallelising the algorithm are not affected. However, by making use of the ability of transputers to overlap communications on different links and to overlap communications and computations, significant speed-ups can be achieved.

On the T800, it is possible to run all four links and the arithmetic unit simultaneously, but due to bottlenecks on the dynamic memory access, full efficiency can not be expected when running more than three processes simultaneously, e.g. three links or two links and the arithmetic unit. While detailed tests would be necessary to determine exactly how much the performance is degraded from the theoretical maximum when, for example, sending data along three links and performing computations simultaneously, models using three and four processors can be developed to estimate the bounds for gains that can be made using multiple transputers. The success of the models developed above in predicting the behaviour of the pipe and the mesh, particularly the latter, suggests that such an analysis will produce reliable results.

First, take the case in which $T_m > T_c$. One example of a three process model is that of a master processor, which performs no computations, connected externally via one link, and to three worker processors on the other three links. Data can be sent simultaneously to all three workers, from the master, while the workers can perform computations on the first data packets received while subsequent packets arrive. Ignoring overheads, (primarily the times of the computations performed by workers after they have received their last packets of data) the total elapsed time for the calculation is $T_m/3$, and hence there will be a gain if $T_m/3 < T_c$, i.e. this model requires

$$T_c < T_m < 3T_c \tag{5.20}$$

Another three process model which will always show a gain is one in which the master processor performs a proportion β of the computations while simultaneously sending data to two workers which each perform a proportion γ of the

computations. Load balancing requires $\beta T_c = \gamma T_m$, which produces

$$\beta = \frac{k}{2+k} \tag{5.21}$$

since $\beta + 2\gamma = 1$, where $k = T_m/T_c > 1$. A four process model would have the master performing δ of the computation and each of three workers ϵ, which gives

$$\delta = \frac{k}{3+k} \tag{5.22}$$

Since $\beta < k/3$ for all $k > 1$, the second and third models both predict shorter total elapsed times for the calculation than the first, which can therefore be ignored. If $k = 1.38$, as for the double precision calculation (table 5.1), then $\beta = 0.41$ and $\delta = 0.32$, and hence it should be possible to reduce the calculation time to roughly 1/3 of that on a single processor, a significant gain.

The case with $T_c > T_m$ is considerably more complicated, since maximum efficiency requires more than one layer of worker processors. Taking first a three process model, then pipes of worker processors can be hung off each of the links on the master. The length of each pipe will depend on $k_1 = T_c/T_m$. Suppose the time taken to send all the data from the master to the first processor in a pipe is T_p, then the time to perform all computations on this data on a single processor would be $k_1 T_p$, and to balance the load on the first processor requires that it performs a proportion, β_l, of the computations where $\beta_l = 1/k_1$. Thus, if $1 < k_1 \leq 2$, then at least half the computations can be performed on the first processor in the chain, and the rest on a second processor. If $2 < k_1 \leq 3$ then the chain would need to be three deep to obtain maximum efficiency. Here, in addition to ignoring the overheads, it has been assumed that it is possible to synchronise the data transfer and computations along the chain in such a way that near maximum efficiency is

achieved. The latter may be difficult without some complicated coding and data management, but the relative high efficiency of Occam with 'short' messages, due to low latency, should help.

Full three process models can now be constructed. With a master processor which does not perform any computations and three chains, total elapsed time is $T_m/3$, i.e.

$$T = \frac{T_c}{3k_l} \tag{5.23}$$

A second three process model with two chains and the master performing a proportion of the calculation can also be constructed, but since its estimated time is worse than that of equation (5.23) no details are given. A four process model is possible with a group of processors on each of three links, and one external link with a proportion β_2 of the computations performed on the master processor. The detailed configuration (and total number of processors) will depend on k_1, but the total time is easily estimated. If each group of workers performs a proportion γ_2 of the total computation, then $\beta_2 + 3\gamma_2 = 1$, and since load balancing requires $\beta_2 T_c = \gamma_2 T_m$, $\beta_2 = 1/(1 + 3k_l)$ is obtained and the total elapsed time is

$$T = \frac{T_c}{1 + 3k_1} \tag{5.24}$$

Note that the time predicted for the four process model (equation (5.24)) is always less than that for the best three process model (equation (5.23)). For $k_1 = 1.27$, as for the single precision calculation (equation (5.23)) gives $0.26T_c$ and equation (5.24) gives $0.21T_c$. The models suggest that it may be possible to reduce the total time for calculation to a quarter of that for a single processor.

5.4 Conclusions

Dynamic Matrix Control, in processing terms, can often be reduced to matrix vector multiplication. The work discussed in this chapter has been concerned with developing cost effective (a limited number of processors available) parallel solutions to the computations involved. Considerable effort has been directed at developing methods of assessing the effectiveness and performance of the parallel architectures.

The T800 transputers have a comparatively (with other commercially available processors) high communication to computation rate, and yet, little speed up was achieved even with problems that provided the most efficient communications rate (where message lengths are all close to the optimum). This would suggest that matrix vector multiplication is not suited to parallelisation. Problems that are most suited to parallelisation are those which have a high computation to communications ratio, i.e problems which require a minimum amount of information to be communicated to them, while a large amount of computation is carried out from that information. Following this, extra speed up would be achieved if it was possible to construct matrix elements from minimal data (e.g. when dealing with sparse or symmetrical matrices). Another route might lie in a transputer specific implementation which takes advantage of the ability to overlap communications and computations; or on machines where data can be loaded directly to multiple processors in the array (say the boundary processors).

With processor speeds increasing at least as fast as communication speeds, it is not easy to envisage future processor arrays providing more speed up on the architectures presented in this chapter. This conclusion is important since it highlights that the DMC problem is not a practical problem for parallelisation on

commodity processors.

Chapter 6

Implementation of Multiple Model Based Adaptive Control Schemes - The Linear Model Case

6.1 Introduction

This chapter describes the development of a parallel processing platform for the implementation of a class of multiple model adaptive control schemes which are known to yield superior performance over alternatives in cases of practical interest. This scheme, using multiple models, switching and tuning, is described in section 6.3. Certain necessary alterations and additions to the original version are then discussed before the scheme is parallelised and fully analysed. Finally, a brief overview of certain optimization strategies is presented. First it is necessary to expand on the MRAC approach which was described in general terms in section 2.2.2.

Conceptually, (Middleton *et al.* 1988) can probably claim the first (renewed)

investigation into this area. The aim is to improve adaptive control systems by increasing the domain 'covered' by the adaptive system. However, the limitations of adaptive control to slowly time-varying plants remains.

There has been continued interest in multiple model switching schemes. - (Morse *et al.* 1992) introduced hysteresis into the switching scheme of Middleton's multiple adaptive model scheme. Morse had earlier shown that multiple model schemes had limited capabilities without hysteresis switching. Hysteresis switching has been applied to multi-variable control by Weller and Goodwin (1994). The systems are limited to those with the same number of inputs as outputs. The process still involves parallel adaptive models. It was proved that all switching stops after a finite time (due to hysteresis) however, this is only applicable to the identification of time invariant plants.

Narendra *et al.* (1995) suggested that instead of distributing parallel adaptive models, a single adaptive model should be run alongside several fixed models distributed across a known parameter space. This method is described in greater detail in section 6.3.1. Note also that this approach also raises the possibility of achieving 'high performance' control of 'fast-varying' systems with the possibility of large discontinuous changes in the plant dynamics. A proof of closed loop stability of the scheme is presented in (Narendra and Balakrishnan 1997) which revolves around the boundedness of inputs and states of the system. A final novel approach to the field has been in the adaptive control of overmodelled plants (Kreisselmeier and Lozano 1996). Over modelling of plants is a problem since there will exist uncontrollable modes if exact matches of the input output mappings of the plant are achieved. The solution presented is to run adaptive models of different order (from the lowest possible order to the highest) in parallel. On the whole, smooth adaptation can be achieved.

6.2 Model Reference Adaptive Control [MRAC]

As discussed in section 2.2.2, MRAC is an area for which there exists a vast literature on both theory and applications. In this chapter one particular form of MRAC is considered for which the following is the necessary theoretical background. The results given are essentially from (Sastry and Bodson 1989) and the relevant cited references and before introducing the MRAC scheme considered, it is necessary to give some basic ideas/results from so-called indirect adaptive control.

There are two main approaches to the design of adaptive control schemes: direct and indirect. Direct adaptive controllers use a direct update law for the controller parameters, whereas the control action of indirect adaptive controllers is divided into two steps. In particular, parameters are first estimated and then used to select the controller parameters. Here an indirect approach must be chosen to derive a model of the plant, which can then be used in the overall control strategy considered there.

The basic structure of an indirect adaptive controller is shown in figure 6.1. The plant is assumed to have a known structure, as defined below, although its parameters are unknown.

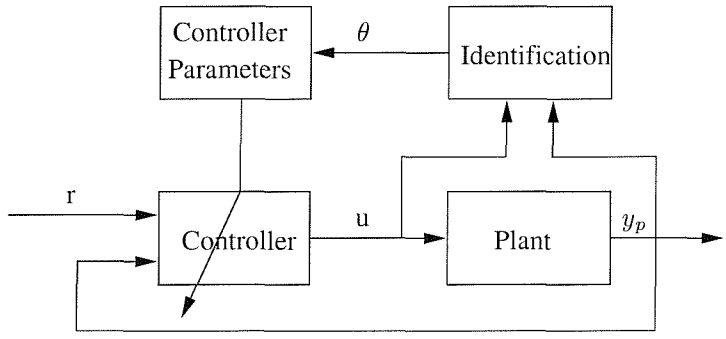


Figure 6.1: The basic structure of an indirect adaptive controller

The controller is parameterised by a number of adjustable parameters. When

the plant parameters are exactly known, the corresponding controller output $u(t)$ should force the plant output $y_p(t)$ to exactly meet the control objectives. When the plant parameters are not known, the adaptation mechanism will adjust the controller parameters in such a way that the control objective is asymptotically achieved. Existing adaptive control designs normally require that the control law is linear in terms of the adjustable parameters in order to obtain adaptation mechanisms with guaranteed stability and parameter convergence.

As noted above, the adaptation mechanism of an indirect adaptive controller is divided into two parts: the identification of a plant model and the derivation of the controller parameters. The adaptation law of the identifier searches for model parameters Θ such that the model output becomes the same as the plant output when the same input $u(t)$ is applied. From these estimates Θ of the plant parameters the controller parameters are derived according to the chosen control strategy, by MRAC here but alternatives exist such as pole placement (Elliott *et al.* 1985). Clearly, the main difference from conventional control lies in the existence of this adaptation mechanism. The main issue in adaptation design is to synthesise an adaptation mechanism which will guarantee that the control system remains stable and fulfills the objective of the chosen control strategy. Many formalisms can be used to this end, such as the Lyapunov theory and hyperstability theory which can, for example, be found in (Föllinger 1993a) and (Föllinger 1993b).Next the necessary background for controller design is given where, in keeping with the two step control action, the description is divided into two parts, starting with the identification of the plant model. Following this, the derivation of the controller (by MRAC) parameters is described.

6.2.1 Identification of the plant

The identification strategy for the continuous time linear SISO systems considered here is now summarised (adapted from (Autenreith 1996)). The task is to estimate the parameters of a plant that can be represented by the transfer function description

$$\frac{Y_p(s)}{U(s)} = G(s) = k_p \frac{B(s)}{A(s)} \tag{6.1}$$

where $Y_p(s)$ and $U(s)$ denote the Laplace transforms of the output $y_p(t)$ and the input $u(t)$ of the plant, and $A(s)$ and $B(s)$ are two monic, coprime polynomials of degrees n and m respectively. Also the plant is assumed to be strictly proper, i.e. $m \leq n-1$, and minimum phase, i.e. no right-half plane zeros. Note also that the plant is not assumed to be stable and the sign of the so-called high frequency gain k_p is assumed to be known. No loss of generality arises from assuming that $k_p > 0$. Finally, the input $u(t)$ is assumed to be piecewise continuous for $t > 0$.

In this work the so-called (Ljung and Söderström (1983)) equation error identification structure is used. The plant transfer function here can be explicitly written as

$$G(s) = \frac{\alpha_n s^{n-1} + \cdots + \alpha_1}{s^n + \beta_n s^{n-1} + \cdots + \beta_1} \tag{6.2}$$

where the $2n$ coefficients $\alpha_1, \cdots, \alpha_n$ and $\beta_1, \cdots, \beta_n$ are unknown. This is a parameterisation of the unknown plant, i.e. a model in which only a finite number of parameters are to be determined.

For identification purposes, it is convenient to find an expression which depends

linearly on the unknown parameters. For example, the expression

$$s^n Y_p(s) = (\alpha_n s^{n-1} + \cdots + \alpha_1)U(s) - (\beta_n s^{n-1} + \cdots + \beta_1)Y_p(s) \tag{6.3}$$

is linear in the parameters α_i and β_i but would require explicit differentiations if it were to be implemented. To avoid this problem, introduce the arbitrary monic Hurwitz polynomial

$$\tilde{\Lambda}(s) = s^n + \lambda_n s^{n-1} + \cdots + \lambda_1 \tag{6.4}$$

Then, using (6.1),

$$\tilde{\Lambda}(s)Y_p(s) = k_p B(s)U(s) + (\tilde{\Lambda}(s) - A(s))Y_p(s) \tag{6.5}$$

and hence use of (6.2) gives the following new representation of the plant:

$$Y_p(s) = \frac{a^*(s)}{\tilde{\Lambda}(s)}U(s) + \frac{b^*(s)}{\tilde{\Lambda}(s)}Y_p(s) \tag{6.6}$$

where

$$\begin{aligned} a^*(s) &= \alpha_n s^{n-1} + \cdots + \alpha_1 = k_p B(s) \\ b^*(s) &= (\lambda_n - \beta_n)s^{n-1} + \cdots + (\lambda_1 - \beta_1) = \tilde{\Lambda}(s) - A(s) \end{aligned} \tag{6.7}$$

The transfer function from $U(s) \rightarrow Y_p(s)$ is given by

$$\frac{Y_p(s)}{U(s)} = \frac{a^*(s)}{\tilde{\Lambda}(s) - b^*(s)} \tag{6.8}$$

and it is easy to verify that this transfer function is $G(s)$ when $a^*(s)$ and $b^*(s)$ are given by (6.7). Also the assumption that $A(s)$ and $B(s)$ are coprime guarantees that this choice is unique. In effect, the output of the plant can now be calculated

without explicit differentiation of $y_p(t)$.

A state space realisation of the above transfer function representation can be obtained by choosing the $n \times n$ matrix Λ_{mat} and the $n \times 1$ column vector b_λ in controllable canonical form, i.e.

$$\Lambda_{\text{mat}} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \ddots & 0 \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & 1 \\ -\lambda_1 & & \cdots & & -\lambda_n \end{bmatrix} \quad b_\lambda = \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \end{bmatrix} \tag{6.9}$$

From which it easily follows that

$$(sI - \tilde{\Lambda}_{mat})^{-1} b_\lambda = \frac{1}{\tilde{\Lambda}(s)} \begin{bmatrix} 1 \\ s \\ \vdots \\ s^{n-1} \end{bmatrix} \tag{6.10}$$

Now introduce

$$\begin{aligned} a_*^T &= [\alpha_1, \cdots, \alpha_n] \\ b_*^T &= [\lambda_1 - \beta_1, \cdots, \lambda_n - \beta_n] \end{aligned} \tag{6.11}$$

and the $n \times 1$ column vectors $w_p^1(t)$ and $w_p^2(t)$ as

$$\begin{aligned} \dot{w}_p^1(t) &= \Lambda_{\text{mat}} w_p^1(t) + b_\lambda u(t) \\ \dot{w}_p^2(t) &= \Lambda_{\text{mat}} w_p^2(t) + b_\lambda y_p(t) \end{aligned} \tag{6.12}$$

with initial conditions $w_p^1(0)$ and $w_p^2(0)$. Then in Laplace transform terms

$$\begin{aligned} \hat{w}_p^1(s) &= (sI - \Lambda_{\text{mat}})^{-1} b_\lambda U(s) + (sI - \Lambda_{\text{mat}})^{-1} w_p^1(0) \\ \hat{w}_p^2(s) &= (sI - \lambda_{\text{mat}})^{-1} Y_p(s) + (sI - \Lambda_{\text{mat}})^{-1} w_p^2(0) \end{aligned} \tag{6.13}$$

and with this notation, the plant description (6.6) becomes

$$Y_p(s) = a_*^T w_p^1(s) + b_*^T w_p^2(s) \tag{6.14}$$

The plant parameters here are constant and hence this last equation also holds in the time domain, i.e.

$$y_p(t) = a_*^T w_p^1(t) + b_*^T w_p^2(t) := \theta_*^T w_p(t) \tag{6.15}$$

where

$$\theta_*^T := \left[a_*^T, b_*^T\right] \in R^{2n} \tag{6.16}$$

$$w_p^T(t)^T := \left[w^{1^T}{}_p(t), w^{2^T}{}_p(t)\right] \in R^{2n} \tag{6.17}$$

These last four equations define a realisation of the new parameterisation where $w_p(t)$ is the generalised state vector for the plant. This has dimension $2n$ and hence this realisation is not minimal but the unobservable modes are those of $\tilde{\Lambda}(s)$ and are all stable.

The vector θ_* is a vector of unknown parameters which is linearly related to the original parameters α_i, β_i through (6.11)-(6.17). Knowledge of one set is equivalent to knowledge of the other. Also in the last form, the plant output depends linearly on the the unknown parameters and hence standard identification algorithms can be used.

In essence, the purpose of the identifier is to produce a recursive estimate $\theta(t)$ of the nominal parameter θ_*. Since $u(t)$ and $y_p(t)$ are available, define the observer

$$\begin{aligned} \dot{w}^1(t) &= \Lambda_{\text{mat}} w^1(t) + b_\lambda u(t) \\ \dot{w}^2(t) &= \Lambda_{\text{mat}} w^2(t) + b_\lambda y_p(t) \end{aligned} \tag{6.18}$$

to reconstruct the plant states where the initial conditions in this observer are arbitrary. Also define the identifier signals

$$\begin{aligned} \theta^T(t) &:= \left[a_*^T(t), b_*^T(t)\right] \in R^{2n} \\ w^T(t) &:= \left[w^{1^T}(t), w^{2^T}(t)\right] \in R^{2n} \end{aligned} \tag{6.19}$$

It now follows that the observer error $w(t) - w_p(t)$ decays exponentially to zero for even unstable plants. Also $w_p(t)$ is such that it can be reconstructed from available signals without knowledge of the plant parameters. The plant output can be written as

$$y_p(t) = \theta_*^T w(t) + \epsilon(t) \tag{6.20}$$

where $\epsilon(t)$ denotes the presence of an additive exponentially decaying term given by

$$\epsilon(t) = \theta_*^T (w_p(t) - w(t)) \tag{6.21}$$

and it is due to the initial conditions in the observer. Also it is possible to neglect the presence of this term since it does not affect the properties of the identifier.

The identifier output is defined to be

$$y_i(t) = \theta^T(t)w(t) \tag{6.22}$$

and the parameter error is defined as

$$\psi(t) := \theta(t) - \theta_*(t) \in R^{2n} \tag{6.23}$$

Also the identifier error is defined as

$$e_i(t) := y_i(t) - y_p(t) = \psi^T(t)w(t) + \epsilon(t) \tag{6.24}$$

These signals are used by the identifier algorithm.

Many identification algorithms rely on a linear expression of the form detailed above, i.e. $y_p(t) = \theta_*^T w(t)$ where it is only the so-called regressor vector θ_* that is unknown. Associated with $y_p(t)$ here is the linear error equation $e_i(t) = \psi^T(t)w(t)$. In effect, here the identifier has been separated into an identifier structure and an identification algorithm. The identifier structure constructs the regressor $w(t)$ and the other signals related by the identifier error equation. Also the identification algorithm is defined by a differential equation - termed the update law - of the form

$$\dot{\theta}(t) = \dot{\psi}(t) = F(y_p(t), e_i(t), \theta(t), w(t)) \tag{6.25}$$

where F is a causal operator which is explicitly independent of $\theta_*(t)$, and defines the evolution of the identifier vector $\theta(t)$.

The Least-Squares algorithm is one of the update laws that can be used for this purpose and the essential steps in this algorithm are now summarised. A

detailed treatment can, for example, be found in (Sastry and Bodson 1989). The first key step in the Least-Squares algorithm is that the derivative of the vector $\Theta(t)$ develops as a function of the output error. Based on this the normalised Least-Squares algorithm is given by

$$\dot{\theta}(t) = -g\frac{P(t)w(t)\epsilon(t)}{1+w^T(t)P(t)w(t)} \tag{6.26}$$

$$\dot{P}(t) = -g\frac{P(t)w(t)w^T(t)P(t)}{1+w^T(t)P(t)w(t)} \tag{6.27}$$

where $P(t)$ is the so-called covariance matrix and g is a constant positive gain to be selected.

The elements of the covariance matrix $P(t)$ can be interpreted as update gains for each parameter and must be initialised, i.e. starting values for its entries must be specified. A commonly employed method is to set $P(t_0)$ equal to a diagonal matrix with non-zero elements in the range $1000 - 10000$. A problem that can arise during adaptation is that the elements of $P(t)$ are decreasing and hence the adaptation process becomes very slow when the gains for the adaptation tend to zero. Numerous, essentially ad-hoc, solutions to this problem have been proposed. Commonly used ones include covariance resetting or the use of a constant trace algorithm (Goodwin and Mayne 1987). In the results in this chapter the former has been employed to emulate the results produced in (Narendra *et al.* 1995). It consists of resetting the entries in the covariance matrix to their initial values when its trace is less than a certain prespecified bound.

Once the vector $\theta(t)$ containing the parameters a_* and b_* is estimated, the plant parameters α_i and β_i are calculated using (6.11).

6.2.2 Control of the Plant

When an estimated model of the plant is known, a deterministic strategy can be used to derive an appropriate controller. There are many control strategies that can be used, such as pole placement or model reference control. In the following, a version of the latter approach will be described.

Derivation of the controller

In keeping with the MRAC strategy, the reference model is represented by the transfer function

$$M(s) = k_m \frac{Q(s)}{P(s)} \tag{6.28}$$

where the polynomials $P(s)$ and Q(s) have the same degrees as $A(s)$ and $B(s)$ respectively. Since an unstable reference model makes no sense $P(s)$ must be a Hurwitz polynomial.

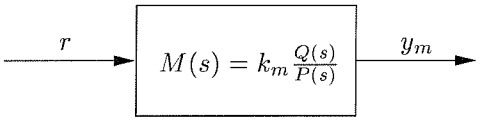


Figure 6.2: The reference model

The objective of MRAC can be achieved by using a controller consisting of three parts, a feedforward gain g, a cascade compensator $\frac{K_{1n}(s)}{K_{1d}(s)}$ and a feedback compensator $\frac{K_{2n}(s)}{K_{2d}(s)}$ as shown in figure 6.3. The closed-loop transfer function of the whole system is

$$S(s) = \frac{Y_p(s)}{R(s)} = \frac{g k_p \frac{B(s)K_{1n}(s)}{A(s)K_{1d}(s)}}{1 + k_p \frac{B(s)K_{1n}(s)K_{2n}(s)}{A(s)K_{1d}(s)K_{2d}(s)}} \tag{6.29}$$

where $R(s)$ is the Laplace transformation of $r(t)$. To simplify the overall scheme,

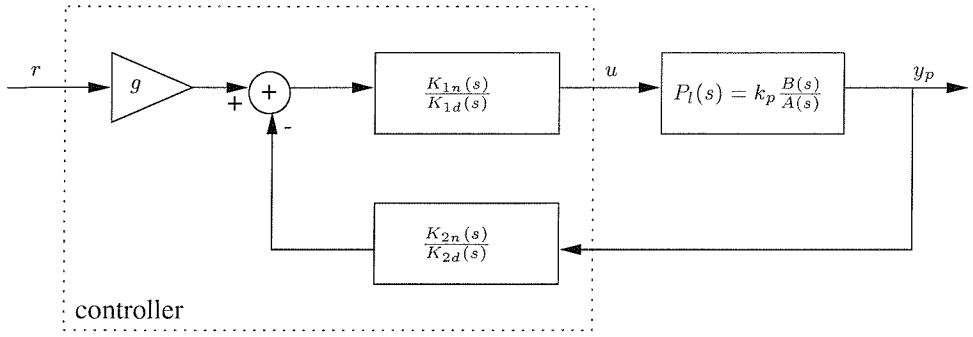


Figure 6.3: The basic control structure

the choice of $K_{1n}(s) = K_{2d}(s) = \Lambda(s)$ is made. It is necessary to choose $\Lambda(s)$ as a Hurwitz polynomial in order to ensure that common roots can be cancelled without the introduction of undesirable effects[1]. Without loss of generality $\Lambda(s)$ can be chosen as a monic polynomial. To match $S(s)$ and $M(s)$, the numerator of $S(s)$ must contain $Q(s)$. Thus a choice of

$$\Lambda(s) = \bar{\Lambda}(s)Q(s) \tag{6.30}$$

is made and, to ensure that $\Lambda(s)$ is a Hurwitz polynomial, $\bar{\Lambda}(s)$ and $Q(s)$ must also be Hurwitz polynomials. These requirements lead to

$$S(s) = k_m \frac{Q(s)}{P(s)} \frac{g\frac{k_p}{k_m}B(s)\bar{\Lambda}(s)P(s)}{A(s)K_{1d}(s) + k_p B(s)K_{2n}(s)} \tag{6.31}$$

The last factor of this equation must the identity. To achieve this, the controller parameters are defined as

$$g = \frac{k_m}{k_p} \qquad K_{1d}(s) = B(s)T(s) \qquad K_{2n}(s) = \frac{1}{k_p}R(s) \tag{6.32}$$

where $T(s)$ and $R(s)$ are two polynomials still to be determined. Now (6.29)

[1]Pole-zero cancellations of unstable roots do not take into account possible internal instabilities

becomes

$$S(s) = \frac{B(s)\bar{\Lambda}(s)P(s)}{B(s)A(s)\Lambda(s) + B(s)R(s)} \tag{6.33}$$

and the polynomial $B(s)$ can be cancelled if it is a Hurwitz polynomial. Finally, the controller polynomials can be calculated from the equation

$$\bar{\Lambda}(s)P(s) = A(s)T(s) + R(s) \tag{6.34}$$

Using this last result, the cascade and the feedback compensator are given respectively by:

$$\frac{K_{1n}(s)}{K_{1d}(s)} = \frac{\Lambda}{B(s)T(s)} \qquad \frac{K_{2n}(s)}{K_{2d}(s)} = \frac{1}{k_p}\frac{R(s)}{\Lambda(s)} \tag{6.35}$$

It is now necessary to consider the degrees of the controller polynomials in order to complete its specification. Consequently, the orders of $\Lambda(s)$ and $\bar{\Lambda}(s)$ are defined to be l and $\bar{l}$ respectively. Then it follows from (6.30) that $l = \bar{l} + m$. Equation (6.34) can be viewed as a polynomial division of $\bar{\Lambda}(s)P(s)$ by $A(s)$. This means that the quotient $T(s)$ has degree $\bar{l}$ and the remainder $R(s)$ has degree $n-1$. Since only proper transfer functions (i.e. $m \leq n$) can be implemented, it follows from the feedback compensator's transfer function that $l \geq n-1$. To ensure the simplest possible controller, $l = n-1$ is chosen and this implies that $\bar{l} = n - m - 1$.

The cascade compensator of (6.35) can be unstable. To ensure a stable transfer function, write

$$\frac{K_{1n}(s)}{K_{1d}(s)} = \frac{1}{1 - \frac{C(s)}{\Lambda(s)}} \tag{6.36}$$

where $C(s) = \Lambda(s) - B(s)T(s)$. This expression corresponds to a feedback loop with a compensator $\frac{C(s)}{\Lambda(s)}$ that replaces the cascade compensator in the above analysis. Since $C(s)$ is the difference between two monic polynomials of order $n-1$, its degree is $n-2$ and hence, the new feedback compensator is strictly proper and stable. Figure 6.4 shows the structure of the final controller of the MRAC scheme. The gain g and the polynomials $\Lambda(s)$ and $C(s)$ are those from above and $D(s) = -\frac{1}{k_p}R(s)$.

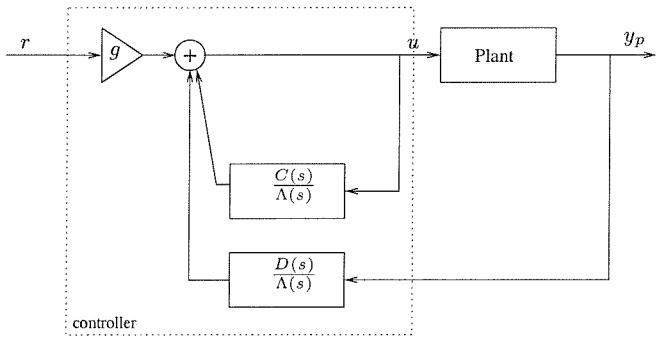


Figure 6.4: A Model Reference Controller

Calculation of the Controller's Output

The above controller can be transformed into another structure which has the advantage that the command $u(t)$ is expressed as a linear combination of a coefficient vector $\Pi(t)$ and a regression vector $w_c(t)$.

By analogy with (6.9) and (6.10) the output of the strictly proper compensator $\frac{C(s)}{\Lambda(s)}$ can be written as

$$Y_u(s) = \pi_c^T w_u' \tag{6.37}$$

where the vector π_c contains the coefficients of $C(s)$ and the regression vector w_u' is obtained by a stable filtering of $u(t)$ with a filter of the form (6.9). The only difference here is that the dimensions of the matrix and the vector are reduced

from n to $n-1$.

The transfer function $\frac{D(s)}{\Lambda(s)}$ is not strictly proper, but can be written as

$$\frac{d_0 s^{n-1} + d_1 s^{n-2} + \cdots + d_{n-2} s + d_{n-1}}{\Lambda(s)} = \tilde{d}_0 + \frac{\tilde{d}_1 s^{n-2} + \cdots + \tilde{d}_{n-2} s + \tilde{d}_{n-1}}{\Lambda(s)} \tag{6.38}$$

The first term on the right hand side of this equation represents a direct feed-through of the output $y_p(t)$. The second term is a strictly proper transfer function with an output given by

$$Y_{y_p}(s) = \pi_{\tilde{d}}^T \overset{\prime}{w}_{y_p} \tag{6.39}$$

where $\pi_{\tilde{d}}$ contains the coefficients $\tilde{d}_1, \ldots, \tilde{d}_{n-1}$ and $\overset{\prime}{w}_{y_p}$ is obtained by a stable filtering operation on the plant output.

From figure 6.4 it follows that the output $u(t)$ of the model reference controller can be calculated as

$$u(t) = g\, r(t) + \frac{C(s)}{\Lambda(s)} u(t) + \frac{D(s)}{\Lambda(s)} y_p(t) \tag{6.40}$$

Using the vectors

$$\pi_c^T = [c_0, c_1, \ldots, c_{n-3}, c_{n-2}] \tag{6.41}$$

and

$$\pi_{\tilde{d}}^T = [\tilde{d}_1, \tilde{d}_2, \ldots, \tilde{d}_{n-2}, \tilde{d}_{n-1}] \tag{6.42}$$

enables two new vectors $\Pi(t)$ and $w_c(t)$ to be defined as:

$$\Pi(t) = \begin{bmatrix} g \\ \pi_c \\ \tilde{d}_0 \\ \pi_{\tilde{d}} \end{bmatrix} \qquad w_c(t) = \begin{bmatrix} r(t) \\ \overset{\prime}{w}_u(t) \\ y_p(t) \\ \overset{\prime}{w}_{y_p}(t) \end{bmatrix} \tag{6.43}$$

Now (6.40) can be written as

$$u(t) = \Pi^T(t) w_c(t) \tag{6.44}$$

These definitions enable the coefficients and the states of the model reference controller to be decoupled.

6.3 Switching Scheme

This section describes a scheme for an adaptive controller for dynamic systems, presented in (Narendra *et al.* 1995). This scheme is particularly novel since, conventionally, adaptive controllers have only been applied to slow-time-varying systems with no large discontinuous changes. Such systems allow the tuning process for the adaptive model to 'keep up' with the changes.

6.3.1 A Scheme for an Adaptive Controller for Discontinuous Time-varying Plants

The scheme involves distributing a number of fixed models across a known parameter space to aid an adaptive controller in applications where the plant parameters are prone to large discontinuous changes; or where external disturbances that ef-

fect the plant dynamics can change suddenly. These fixed models operate along side the adaptive model and are evaluated using a cost function based on their identification errors. This cost function is defined as:

$$J_i(t) = \alpha\epsilon_i^2(t) + \beta\int_0^t e^{-\lambda(t-\tau)}\epsilon_i^2(\tau)d\tau \tag{6.45}$$

where the positive constants α and β dictate the relative importance of the current error ($\epsilon_i(t)$) and past errors ($\epsilon_i(\tau)$) respectively, and λ affects the length of 'memory' of the second term. The controller corresponding to the model with the minimum cost function is then switched to with a certain hysteresis to prevent arbitrarily fast switching between similar models. That is if J_{new} is the minimum cost function, J_{cur} is the cost function of the model corresponding to the current controller, and δ is the hysteresis constant; then a switch will occur if:

$$J_{new} + \delta < J_{cur} \tag{6.46}$$

Now that there is a method to assess the current best model, a final aid to the free-flowing adaptive model is added: a second resettable adaptive model which is reset to the parameters of the best model at the beginning of each sampling period. The system is shown, diagrammatically, in figure 6.5.

It was decided to distribute the fixed models uniformly across the parameter space. A border of models (one model thick - equivalent to lengthening the parameter space down each side by the distance between each model) was distributed around the parameter space - this improves the performance of the scheme with plants that lie on the edge of the parameter space. The distribution is kept simple by taking the nth root (n being the number of parameters) of the number of global

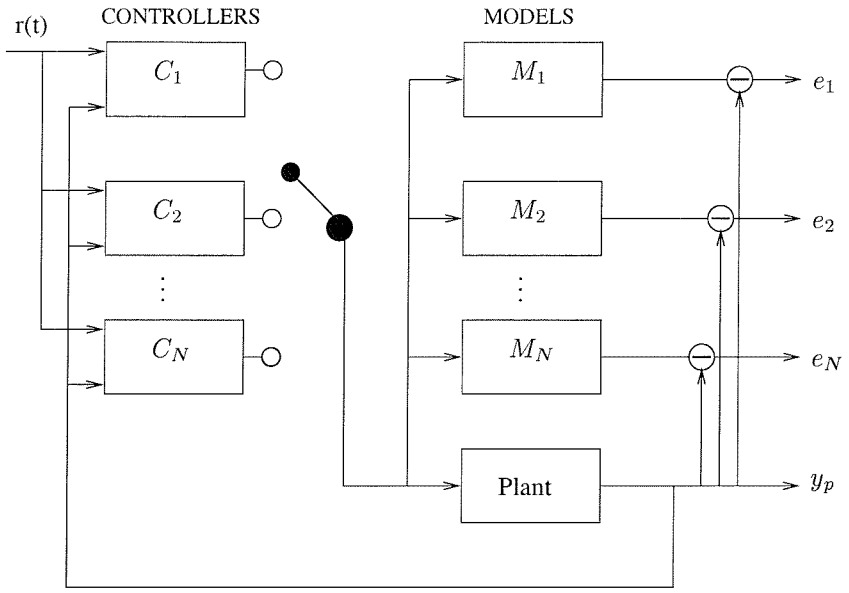


Figure 6.5: Switching scheme involving multiple models, switching and tuning.

fixed models required by the designer. A primitive 'intelligence' is then built in where more models are distributed down the longest side of the parameter space at the expense of those down the shortest side of the parameter space. The aim of this is to make the distance between the models as close to equal as possible.

6.3.2 Necessary Alterations

An investigation into the scheme described in the previous section has been carried out using MATLAB (Autenreith 1996) and, as a result certain alterations have been made to improve performance:

Averaging

The cost functions are used to produce a mean average model of all fixed

models using:

$$\Theta_{av} = \frac{\sum_{n=0}^{N} \frac{1}{j_n(t)} \Theta(t)}{\sum_{n=0}^{N} \frac{1}{j_n(t)}} \tag{6.47}$$

and a corresponding controller is produced from these mean parameters. The underlying switching scheme is still maintained, since it is still necessary to identify the current best model (see (6.47)) for the detection of changing plant dynamics (when a new model is switched to).

Resetting the States

$$y_i(t) = C_i^{A_i(t-t_0)} x_0 + \int_{t_0}^{t} e^{A_i(t-\tau)} B_i u(\tau) d\tau \tag{6.48}$$

Equation (6.48) shows the output calculation for each model (this should be compared with the state-space representation of (2.8) and (2.9)). The scheme works on the assumption that the initial state vector (x_0) term will converge to zero; leaving the model output dependent solely on its parameters. This means that should the parameters of a model match the plant parameters, their outputs will converge towards one another. However, this is only true for stable plants. With unstable plants the initial state vector term becomes the dominant term and even if models share the plant's parameters their outputs will not converge. To remove this problem it is necessary to reset all model states to those of the plant whenever the plant parameters change. As it is not possible to identify when the plant dynamics change directly (it is the job of the scheme to detect these changes when they arise), the states are reset whenever a new model is switched to.

Biased Switching

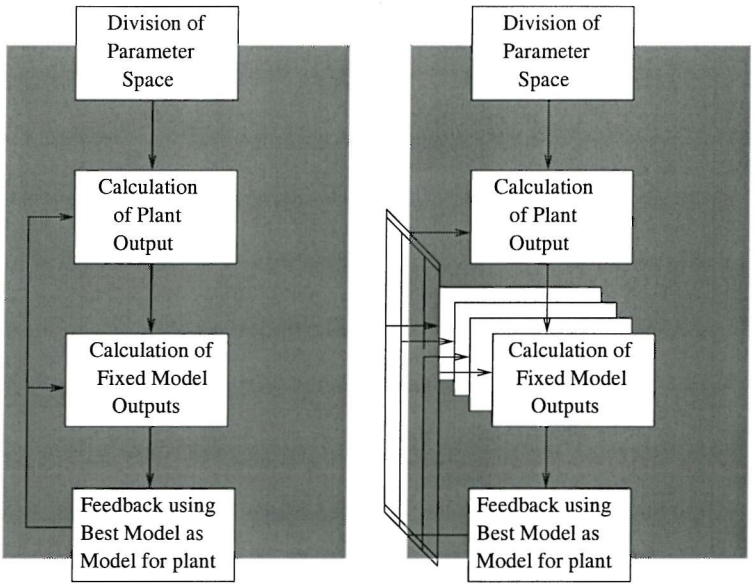
When a large number of fixed models are used in a simulation, rapid switching between models can occur, which causes a continual resetting of states as above. This has the additional effect of not allowing the resettable adaptive model to tune to the plant parameters properly. A solution to this is to remove the hysteresis condition from the adaptive models, forcing the switching scheme to switch to an adaptive model when its cost function becomes the minimum. Once an adaptive model is switched to, the hysteresis constant δ is multiplied by a positive constant ($K > 1$) in an attempt to prevent the scheme from switching to another model (see (6.49)). This affects the detection of changes in the plant dynamics (as described above). However, the first adaptive model to be chosen will be the resettable model, therefore, as soon as the cost function of the free-flowing adaptive model becomes the minimum it is switched to, and the hysteresis constant is reset to its initial value, once more increasing the scheme's sensitivity to changes in plant dynamics.

$$J_{new} + K\delta < J_{cur} \tag{6.49}$$

6.4 Parallelisation of scheme

6.4.1 Parallelisation of the Sequential algorithm

In considering whether parallel processing would lend itself easily to this switching scheme, the sequential process needs to be examined in more detail (see figure 6.6a). This division of the parameter space could be carried out by a master and forwarded to the workers; or, if the parallel harness allows it, the workers can



(a) The Sequential Algorithm (b) The Parallel Algorithm

Figure 6.6: Parallelisation of the sequential algorithm

calculate their own sub-section from global knowledge of the parameter space. The calculation of the model outputs is the most computationally intensive part of each sampling period (involving a Runge-Kutta step to be performed on each model). This also corresponds to the most parallel part of the algorithm. Using parallel processing, not only could the same number of models be processed in a shorter time period, but also, if desired, more models could be processed in the same sampling period, providing a more robust controller than its sequential counterpart. Sandwiching this is an inherently sequential section where the plant output must be sampled and control inputs computed. However, in large simulations, with many fixed models per processor, this sequential component will become less significant. Figure 6.6b shows the parallel equivalent of the flow diagram.

6.4.2 The Topology and Communications Strategy

The parallel platform to be used (for more information see section 3.5) is a network of sixteen T800 transputers, divided into boards of two processors. Each board is connected to a back plane, forming a network that is fully reconfigurable, allowing the modelling of any topology, within the limits of the sixteen nodes. This allows a master processor (which is to be put in charge of simulating the plant, communicating data to other processors and calculation of control inputs) to be connected to all other processors (which are to be used to calculate model outputs). The most notable feature of the transputer is that it can communicate with other transputers, via four two-way links, simultaneously with computation (i.e. at the extreme a transputer could be communicating down all four of its links while performing an unrelated computation).

The processors are connected in a tree structure at the software level, with virtual links connecting the workers to the master (see figure 6.7). Virtual links allow communication to be carried out between processors without a concern for how messages are to be routed through the actual network (at the hardware level). The PARIX implementation (Parsytec 1993) (see 3.4.3 for more information) uses a software router (when dealing with T800's) to emulate a T9000/C104 network when interpreting the virtual links, which aims to produce the optimal routing between processors. The T9000 was the planned upgrade of the T800, and the C104 is a routing chip that can be connected to 32 transputer links.

No communication occurs between workers (as would be expected in a conventional process farm). In order to take full advantage of the parallel components of the scheme, communication has been kept down to three main packets.

Plant Data - from the master to the workers - containing the control input and

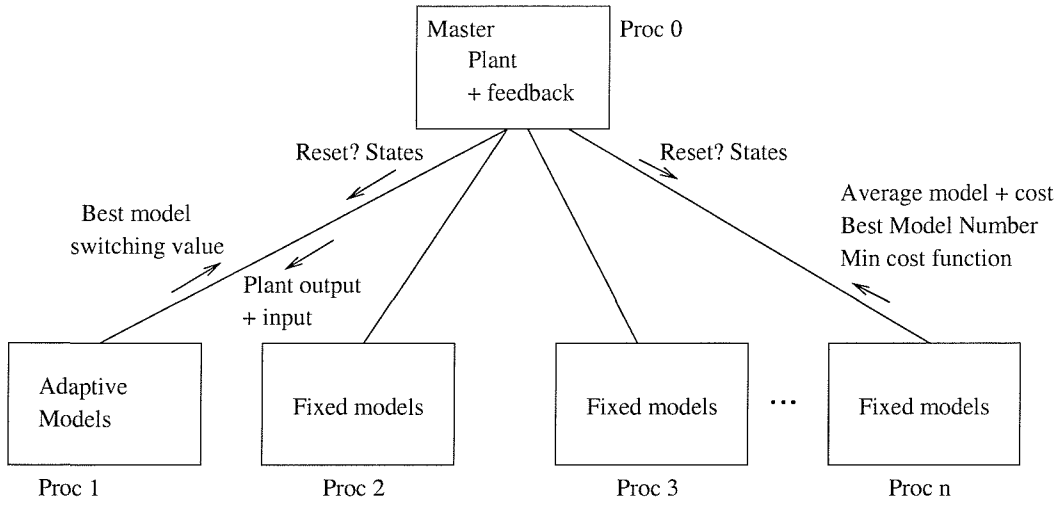


Figure 6.7: Tree topology. One Master processor is connected (via virtual links) to all other processors.

plant output.

Model Data - from the workers to the master - containing the averaged model parameters, an averaged cost function, the best model number, and the minimum cost function (to allow the master to determine if a new model has been switched to under the old scheme). The averaged cost function is simply:

$$\frac{1}{j_{av}} = \frac{\sum_{n=0}^{N} \frac{1}{j_n}}{N} \tag{6.50}$$

Best Model Data - From the master to the resettable adaptive model - containing the best model parameters (whether averaged or adaptive). No effort is made to determine whether the resettable model is the best model.

One further communication is necessary, the master needs to inform the workers whether the states of all models need to be reset, and if so, communicate the states of the plant to all processors.

The master has to perform the top level switching scheme once all commu-

nications are received from the workers. With each worker processor calculating a small subset of the global switching scheme, hysteresis switching is only necessary on the processor from where the best model has come, and so the master communicates the best processor number as well as the best model number to all processors. The worker processors calculate a weighted average of the model parameters (with the weighting provided by each model cost function), the best model and best cost function is also communicated to allow the master to calculate the switching scheme for the purpose of detection of changes in plant dynamics. As the packets are of a fixed size, the larger the simulation (i.e. the more fixed models distributed across the parameter space), the less significant the communication time will become. In fact, the larger the problem, the more significant the computation of model output becomes, and the closer the parallel harness moves to an n-times speed up (n being the number of processors).

6.5 An Example

The following example is a variation of an example first presented in (Narendra *et al.* 1995).

It is known that the plant can be modelled by the transfer function $y_p(s) = \frac{k_p}{s^2+a_1 s+a_0}$ and that the parameters (k_p, a_1, a_0) lie in the parameter spaces: [0.5 2.0], [0.25 2.0], and [-1.0 2.0] respectively. The plant output is to be made to follow the output of the model of reference $R = \frac{1.0}{s^2+1.4s+1.0}$ which accepts a square wave input with a period of 10 time units. The plant dynamics change every 50 time units, in the following sequence:

$$\frac{1.25}{s^2+2.0s+1.0} \rightarrow \frac{0.5}{s^2+2.0s-0.8} \rightarrow \frac{0.5}{s^2+s+1.5} \tag{6.51}$$

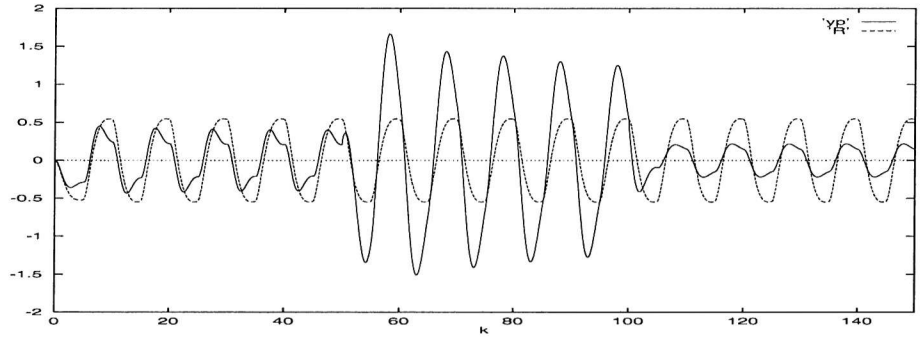
The Covariance matrix, $P(0)$, is initialised to a diagonal matrix with non-zero elements of 10000. A hysteresis constant of $\delta = 0.5$ is used with a biasing of $K = 3$. The following cost function is used for evaluation:

$$J_i(t) = 100\epsilon_i^2(t) + 200\int_0^t e^{-0.5(t-\tau)}\epsilon_i^2(\tau)d\tau \tag{6.52}$$

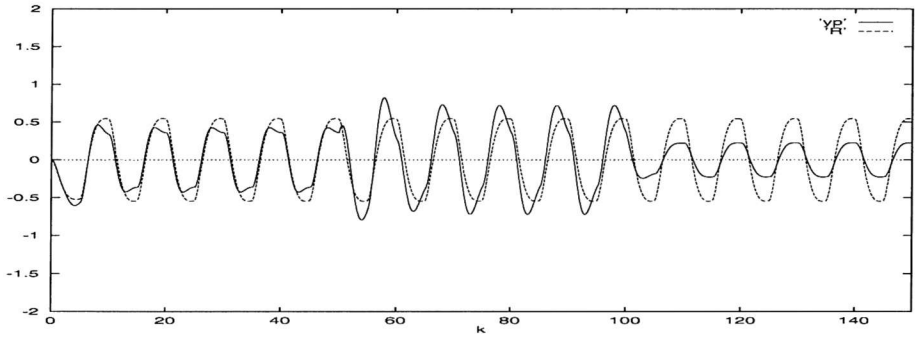
6.5.1 Results and Discussion

Figures 6.8a and 6.8b show the results. The reference model is shown as a dashed line, with the plant output shown as a solid line. Figure 6.8a should be used as a comparison when viewing figure 6.8b and shows control using a single adaptive model and no fixed models. It manages reasonably well with the stable regions. However, when the plant goes unstable (at the 50th sample/time unit) it oscillates badly. The controller is slowly gaining in authority as it approaches a stable region again, but the inaccuracy of control is unlikely to be satisfactory in any practical sense.

Figure 6.8b shows the improvement when 112 fixed models are introduced into the parameter space. The plant is almost immediately forced back into line after the change in dynamics (where the plant becomes unstable). Improvement can also be seen in the stable region, where the reference model is followed more smoothly. In both results, the last region (from 100 to 150 time units) the plant output is slightly overdamped. This is most probably due to the particular choice



(a) A single adaptive model



(b) 112 fixed models distributed across the parameter space

Figure 6.8: results

of design parameters (λ_1 and λ_2 in (6.9)).

6.6 Performance Analysis

A limited performance analysis has been carried out on the example given in the previous section. Table 6.1 shows relative computational times of similar problems (those with the same number of fixed models globally). In the time column, the numbers in the parentheses are projections based on the time achieved by 16 processors given an n-times speed-up. The results show that there is no significant loss of processing time due to communication (i.e. comparing the projection of the three processor problem and the actual result, there is only about a 3% variation)

Number of Processors (fixed models on n - 2)	Processor Partition	Number of fixed models globally (*per processor*)	Average run time (s) *(projected time based on 16 processor result)*
16	[4 x 4]	112 (*8*)	164.55
12	[4 x 3]	110 (*11*)	223.30 (*226.26*)
9	[3 x 3]	108 (*12*)	242.52 (*246.83*)
8	[4 x 2]	114 (*19*)	381.69 (*390.81*)
6	[3 x 2]	112 (*28*)	560.62 (*575.93*)
4	[2 x 2]	112 (*56*)	1118.47 (*1151.85*)
3	[3 x 1]	112 (*112*)	2234.68 (*2303.70*)

Table 6.1: Run times of a similar problem size distributed across a varied number of processors

which is a sign that the problem is scaling well. No attempt has been made to ascertain any loss in performance due to the translation from a sequential program to a parallel one, since no sequential program exists for the transputers. Currently, the three processor partition is the minimum sized problem. Table 6.2 compares problems where the number of fixed models per processor remains constant. Again, there is a good correlation, with a less than 1% variation between results. Both tables demonstrate that this application has good scalability, i.e. there is a linear relation between problems size and the number of worker processors. These results are particularly impressive since the ability of the transputer to overlap communication and computation has not been implemented in this scheme.

It is important to note, however, that there is a practical limit to the scalability of the problem. The analysis assumes a lower limit of fixed models per processor. Also, there exists a practical limit in terms of how effective the algorithm can be to a given problem; i.e. there will be an upper bound of global fixed models, beyond which no extra performance will be gained by adding more models. The existence of a lower limit (or fineness of granularity) requires a little more explanation. One of the reasons why the loss of computational speed-up in the first table 6.1 is greater than that of the second (table 6.2) is almost certainly due to the granularity falling below the perfect level. Since the communication size is constant, the further losses

Number of Processors (fixed models on n - 2)	Processor Partition	Number of fixed models per processor	Average run time (s)
16	[4 x 4]	27	543.44
12	[4 x 3]	27	542.30
9	[3 x 3]	27	541.46
8	[4 x 2]	27	541.19
6	[3 x 2]	27	540.69
4	[2 x 2]	27	540.12
3	[3 x 1]	27	539.87

Table 6.2: Run times across a varied number of processors, keeping the number of fixed models per processor constant.

(the 2%) could be due to the fixed model processors waiting to communicate the results to the master. However, 2% is such a small factor it could just as easily be due to communication bottlenecks in the routing process. There will be a lower limit of granularity (i.e. the point at which the adaptive model calculations become dominant). This will be greater than one since the adaptive model has to perform the Runge Kutta calculation as well as a Least Squares tuning process.

The minimum granularity could be reduced slightly by introducing an improved communication system into the topology. If processors are polled[2] the master would be able to skip over the adaptive model processor, receive all results from the other workers, before returning to the adaptive model processor. However, this strategy would conflict with the idea of moving the adaptive model onto the master processor (see section 6.7), which has advantages in course grain problems.

6.7 Optimisation Potential

The performance analysis was carried out on a computer program on which no attempts at optimisation have been carried out. This section describes areas that

[2]Polling is where a communication times out 'immediately' if no response is received from a communicating partner.

could benefit from optimisation. They are as follows:

Move the adaptive model computation onto the master processor

If the algorithm and the topology diagram are examined (Section 6.4.1 and Figure 6.6), it can be seen that the master processor is idle between the communication of plant data and receiving results from the workers. The first worker will usually (except for very small problems) have less work than the other workers (a Least Squares calculation, compared to Runge-Kutta steps for each fixed model). If the computation is moved onto the master processor, then this will free up another processor for fixed model computation and eliminate some of the idleness of the master.

Load Balancing

Load Balancing is the art of attempting to divide up the computational work between processors equally. However, this could be considered irrelevant in this case since a good choice for the number of fixed models globally will achieve this automatically (Section 6.3).

Topology Optimisation

This describes an area of study concerned with reducing the relative distance between processors that need to communicate. The parallel algorithm has been implemented with out much concern for the actual hardware configuration. It has been assumed that the software router will find the optimal route between communicating processors and that a topology made up of virtual links connecting all worker processors to the master would be sufficient. It is possible that the router might benefit from a more intelligent 'virtual topology'.

A simple approach would be to move the master processor to a central position in the processor partition which would reduce the average distance between pro-

cessors. This makes the assumption that the communication between processor boards varies. Some of the research in this thesis (c.f. section 5.4) has suggested that this is indeed so.

Another approach, which makes the same assumption in relative distance as above, is to create a completely new topology. Several have been suggested as candidates for processor arrays (for example Meshes, Hypercubes, Binary trees - see (Leighton. 1992) - among others) or specific topologies for transputer networks (for example see (Baude 1989)).

Note, however, that this research deals in generic tools, and any topology optimisation will almost certainly make the algorithm architecture dependent.

6.8 Conclusions

This chapter has presented a novel parallel implementation of a multiple model switching scheme for robust adaptive control - first described in (Narendra, Balakrishnan, and Ciliz 1995). It has been demonstrated that the scheme works well under testing conditions where the plant dynamics are allowed to change in comparatively (compared to conventional adaptive controllers) large steps.

The scheme is highly parallelisable and it has been shown that overheads in communication are small making the scheme highly scalable within certain 'natural' boundary determined by the plant to be controlled. There exists an upper limit of fixed models for each problem, above which no further improvement in control will be obtained. There also exists an upper limit of processor scalability, beyond which no further processors can be added to achieve speed-up. The lower limit of granularity is with one model running on each processor with the achieved speed-up of running at a fine granularity perhaps not justifying the added cost

compared to a coarser grain approach. At this granularity the adaptive model will be the dominant process and this could raise the lowest granularity at which speed-up can be achieved. It should be noted that the adaptive model is computationally more intensive than a single fixed model because the adaptive model has to perform a Runge Kutta step and a Least Squares tuning of its parameters; it is not possible to deduce the lowest granularity without first knowing which algorithm is more computationally intensive, the Runge Kutta or the Recursive Least Squares.

Chapter 7

Theory of Neural Network Based Nonlinear Control

7.1 Introduction

The last two decades, in particular, has seen an explosive growth in research (pure and applied) related to neural networks and their applications. During this period the multilayer feedforward network was introduced (Hopfield 1982; Rumelhart *et al.* 1986) and has since been applied in many fields. In particular, it has been particularly successful in areas of the general field of pattern recognition, i.e. static systems. This, in turn, naturally led to interest in its application to dynamic systems and, in particular, control systems.

Research into neural network (and related techniques such as fuzzy logic) based control systems has been a very active research area in recent years with literally thousands of publications in the open literature. One clear fact that has emerged is that such (often termed 'intelligent control') approaches have little extra to offer in the control of a process for which an adequate linear approximate model (or

models) is available. The domain of interest must therefore be nonlinear processes (with or, more likely, without detailed structural knowledge).

In the nonlinear control systems area, numerous models for the practical identification of nonlinear dynamical systems have been reported and later used for the design of controllers. A very large section of this work is of a heuristic nature, i.e. not underpinned by a rigorous theoretical base. Indeed many papers in this area follow a pattern of claiming that 'intelligent control' is far superior to control theory rigorous approaches. A large number of them even claim that 'you simply do not need to know anything at all about the plant dynamics' - all that is needed is input/output data used to train, for example, a neural network to model the plant. Such claims are then supported by 'one-off' empirical (i.e. non-repeatable) designs supported by simulation studies. Even when these techniques are applied to examples (usually from the open literature) where alternative designs are available, there is a marked lack of comparative performance studies.

The facts of the previous paragraph have led to 'intelligent control' in this context being dismissed by large sections of the community at large but has prompted others to attempt to answer the question.

Is it possible to embed 'intelligent control' techniques within a rigorous control theory for nonlinear systems?

The next chapter will implement the schemes which were presented in (Levin and Narendra 1993; Narendra *et al.* 1995; Levin and Narendra 1996), this chapter is concerned with the establishment of some essential theory.

7.2 Background

From a mathematical viewpoint, the control of known nonlinear dynamic systems is a formidable task (for example, all of the benefits of the transfer function/frequency response methods in the linear case are almost always not applicable). This problem becomes substantially more complex when the underlying description of the plant dynamics is only partially known. In such cases, very strong assumptions have to be made if neural network (or related structures such as neuro-fuzzy networks) are to be employed in the identification and control of such plants. The work reported in this chapter is in the spirit of Narendra *et al* - see, for example, (Levin and Narendra 1993; Levin and Narendra 1996) - where the objective is to use and/or develop neural network based theory which can then be used in the practical synthesis of identification based control schemes for partially known nonlinear systems. (For an alternative viewpoint on the rigorous use of neural network/fuzzy logic based techniques in nonlinear control see, for example, the work of French *et al* (French *et al.* 2000; French and Rogers 1998; French and Rogers 1997)).

The particular contribution is on software implementation/processing of the resulting schemes for which the following is the essential theoretical background. The text mostly follows (Levin and Narendra 1993) in presenting these results.

Consider the nonlinear discrete time system state space model

$$\begin{aligned} x(k+1) &= f[x(k), u(k)] \\ y(k) &= h[x(k)] \end{aligned} \tag{7.1}$$

where $x(k) \in R^n$, $y(k) \in R^m$, $u(k) \in R^l$ are the state, output, and control input vectors respectively at sample k. The basic control problem is (as always) to

choose the control input vector $u(k)$ such that the system behaves in a desired manner. It is a well known fact that this problem is 'very challenging' even in the (ideal) case when the nonlinear functions f and h are known. For example, for solutions of (7.1) to exist, f and h must satisfy various conditions which could well be quite involved and difficult to verify (see, for example, (Isidori 1989)).

In adaptive control, f and h are assumed to be unknown and hence the problem is significantly more complex. To obtain a tractable problem, it is necessary to introduce assumptions concerning the controllability and observability properties of the plant/system to be controlled and hence on f and h. Even in the much simpler case of adaptive control of linear time-invariant systems, prior assumptions about the dynamics must be introduced to obtain a solution, eg system order, relative degree and high frequency gain. One means of progressing the nonlinear case is to first introduce similar assumptions to get a 'baseline' solution and then seek to relax them.

The general control problem can be decomposed into the so-called tracking and regulation problems respectively, where in the former the main objective is to stabilize the plant around a fixed operating point. In the latter, the aim is to force the output to follow a specified, or reference or target, signal asymptotically. The most general case is to determine the control input u based only on output measurements (y) for both regulation and tracking. A more restricted, but still highly relevant, version of this problem arises when the state (x) of the system is available and in this case the first equation in (7.1) need only be considered.

The following definitions and results are fundamental in nonlinear control systems theory.

Definition 1 *A point $\hat{x}$ is an equilibrium (or equilibrium state) of $x(k+1) =$*

$f[x(k), u(k)]$ if there exists an input $\hat{u}$ such that $\hat{x} = f[\hat{x}, \hat{u}]$.

Every system considered is assumed to have at least one equilibrium state and, without loss of generality, both $\hat{x}$ and $\hat{u}$ can be chosen to be zero. Hence from this point onwards the origin is considered to be the equilibrium state.

Definition 2 *A dynamical system is said to be controllable if, for any two states x_1 and x_2, there exists an input sequence of finite length that will transfer the system from x_1 to x_2.*

In effect, controllability relates to the ability to influence the state of a dynamical system through the application of inputs. As such, it is a basic concept in systems theory. Also in the nonlinear case, conditions for global controllability are very difficult to establish and verify. Hence attention here is confined to local concepts as in the next definition.

Definition 3 *A system is locally controllable around an equilibrium state $x = 0$ if, for every neighbourhood V of the origin, there is some neighbourhood V' of the origin such that for any two states $x_1, x_2 \in V'$, there exists an input sequence of finite length that will transfer the system state from x_1 to x_2 without leaving V.*

Controllability simply guarantees that a control input vector u exists which can transfer the system from one state to another in a finite number of steps and can either be a function of k or a function of the state at time k. The former case here is termed open loop control, and if $x(k_0) = x_1$ and $x(k_T) = x_2$, the open loop control input $u(k)$ is computed only from a knowledge of x_1, x_2, k_0 and k_T. Since such a control input at $k : k_0 < k < k_T$ is not explicitly determined by the actual state of the system at that instant, it follows that open loop control can be sensitive to noise and external disturbances. The second option - so-called

closed loop control - chooses u as a function of the system state and is robust with respect to such disturbances.

Suppose now that feedback control is used, i.e. $u = g(x)$. Then the system $(x(k+1) = f[x(k), u(k)])$ becomes autonomous and is described by

$$x(k+1) = f[x(k), g(x(k))] = \hat{f}[x(k)] \tag{7.2}$$

The choice of the (state) feedback control law depends on the behaviour expected of the controlled system (7.2).

Definition 4 *Let $x = 0$ be an equilibrium point of (7.2). Then the origin is a stable equilibrium if for every neighbourhood V of the origin there is a neighbourhood $V' \subset V$ of zero such that every solution $x(k)$ with $x(0) \in V'$ is in V for all $k > 0$.*

If V' can be chosen such that, in addition to the properties defined above, $\lim_{k\to\infty} x(k) = 0$, then the origin is asymptotically stable. If this can be achieved in a finite number of steps (n) then V' is finitely (n-step) stable with respect to the origin. When V' equals the whole space, then the origin is globally asymptotically stable.

As a follow on from the definition of stability, the following fundamental system property can be defined.

Definition 5 *If there exists a feedback law that makes an equilibrium point $x = 0$ stable, then the system is stabilizable around that point.*

Some well known theorems from functional analysis are central to the underlying theoretical results considered here. In particular, the inverse function theorem, the explicit function theorem, and the contraction mapping theorem are used. The first two theorems can be found, for example, in (Lang 1983), and in

this context are used to determine the control input explicitly as a function of the state. For convenience, the contraction mapping theorem is stated below for which the following definition is an essential preliminary.

Definition 6 *Let X and Y be normed vector spaces and let $L \subset X$. Consider also an operator $T : L \to Y$. Then if there is a constant c such that*

$$||T(x_1) - T(x_2)|| \leq c||x_1 - x_2||, \text{ for all } x_1, x_2 \in L \tag{7.3}$$

then T is said to satisfy a Lipschitz condition. Also T is a contraction mapping if $c < 1$. (Here $||\cdot||$ denotes the norm on both L and Y.)

Theorem 1 *Let L be a closed subset of the normed vector space X. Let $T : X \to X$ be a contraction mapping on L. Then there exists a fixed point $\overline{x} \in L$ such that $T(\overline{x}) = \overline{x}$. Also for all $x \in L$, $\lim_{k\to\infty} T^k(x) = \overline{x}$.*

Lyapunov theory is a key tool in the stability analysis of dynamical systems and is treated in most advanced control systems texts. The definitions and results required here are given next.

Definition 7 *A function $V(x)$ is said to be positive definite in a region W containing the the origin if:*
(1). $V(0) = 0$, and
(2). $V(x) > 0$ for all $x \in W$, $x \neq 0$.

Definition 8 *Let W be any set in R^n containing the origin and $V : R^n \to R$. Then V is termed a Lyapunov function of the system $x(k+1) = f[x(k)]$ on W if:*
(1) V is continuous on R^n,
(2) V is positive definite with respect to the origin in W,

(3) $\Delta V(k) := V[x(k+1)] - V[x(k)] \leq 0$ *along the trajectories of the system for all* $x \in W$.

The next result shows that the existence of a Lyapunov function guarantees stability.

Theorem 2 *If* V *is a Lyapunov function of the system* $x(k+1) = f[x(k)]$ *in some neighbourhood of the equilibrium state* $x = 0$ *then this equilibrium state is stable. If, in addition,* $-\Delta V$ *is positive definite with respect to* $x = 0$ *then the origin is asymptotically stable.*

These definitions of stability and asymptotic stability are given in terms of perturbations of initial conditions but here a neural network will be used to model a 'real world' process and therefore an exact (or perfect) model will not exist. Instead if the 'real' process is described by $x(k+1) = f[x(k)]$ then the resulting neural network based model will be given by

$$\hat{x}(k+1) = \hat{f}[x(k)] = f[x(k)] + e[x(k)] \tag{7.4}$$

where $e[x(k)] = f[x(k)] - \hat{f}[x(k)]$ is the modelling error and for the system $x(k+1) = f[x(k)]$, $e = e[k, x(k)]$ and depends explicitly on k.

The basic premise here is that if e is 'small', then the behaviour of the original system will be (at least qualitatively) similar to that of the model, i.e. (7.4). To formalise this, the concept of stability under perturbations is required.

Definition 9 *Let* $x(x_0, k)$ *denote a solution of* $x(k+1) = f[x(k)]$ *with initial condition* $x_0 = x(x_0, 0)$. *Then the origin* $x = 0$ *is said to be stable under perturbations if for all* $\epsilon > 0$ *there exists* $\delta_1(\epsilon)$ *and* $\delta_2(\epsilon)$ *such that* $||x_0|| < \delta_1$ *and* $||e(k,x)|| \leq \delta_2$ *for all* $k \geq 0$ *imply* $||x(x_0,k)|| < \epsilon$ *for all* $k \geq 0$. *If, in addition, there is an* r *and a*

$K(\epsilon)$ such that $||x_0|| < r$ and $||e(k,x)|| \leq \delta_2(\epsilon)$ for all $k \geq 0$ imply $||x(x_0,k)|| < \epsilon$ for all $k \geq K(\epsilon)$, the origin is said to be strongly stable under perturbations.

Theorem 3 *If f is Lipschitz continuous in a neighbourhood of the equilibrium, then the system $x(k+1) = f[x(k)]$ is strongly stable under perturbations if, and only if, it is asymptotically stable.*

To introduce the concept of observability, consider (7.1) and define the so-called input and output sequences of length l as

$$\begin{aligned} U_l(k) &= [u(k), u(k+1), \cdots, u(k+l-1)] \\ Y_l(k) &= [y(k), y(k+1), \cdots, y(k+l-1)] \end{aligned} \tag{7.5}$$

Then by the definition of the state, it follows that $x(l+1)$ can be represented as

$$x(k+l) := F_l[x(k), U_l(k)] \tag{7.6}$$

where $F_l : X \times U_l \to X$. Also the output at time $k+l$ can be written as

$$y(k+l) = h[F_l(x(k), U_l(k))] := \overline{h}[x(k), U_l(k)] \tag{7.7}$$

where $\overline{h} : X \times U_l \to Y$. Finally, $Y_l(k)$ can be expressed as

$$Y_l(k) := H_l[x(k), U_{l-1}(k)] \tag{7.8}$$

where $H_l : X \times U_{l-1} \to Y_l$.
When the context is clear, the index k will be omitted, eg $U_l \equiv U_l(k)$.

Observability is now defined as follows.

Definition 10 *A dynamical system is said to be observable if given any two states*

x_1 and x_2 there exists an input sequence of finite length l, i.e. $U_l = (u(0), u(1), \cdots, u(l-1))$, such that $Y_l(x_1, U_l) \neq Y_l(x_2, U_l)$ where Y_l is the output sequence.

Essentially, the ability to effectively estimate the state of a system, or to identify it based on input-output observations, is determined by its observability properties.

Consider now the nth order linear time invariant system described by the state space model

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) \\ y(k) &= Cx(k) \end{aligned} \tag{7.9}$$

Then a basic result in linear systems theory is that this system is observable if, and only if, the so-called observability matrix

$$M_c := \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix} \tag{7.10}$$

is nonsingular (or, equivalently, has rank n).

Observability of a linear system is a system theoretic property and remains unchanged even when inputs are present - provided they are known. If a linear system of order n is observable then any input sequence of length n will distinguish any state from any other state. If two states cannot be distinguished by this randomly chosen input, they cannot be distinguished by any other input sequence. In such a case, i.e. the system is not observable, the system can be realized by an observable system of lower dimension (order).

A single definition of observability is adequate for the linear case only - the concept of observability for nonlinear systems is somewhat more complex. A desirable situation would be that any input sequence of length l would suffice to determine the state uniquely for some integer l. This is known as strong observability and it is easy to show that any observable linear system is strongly observable with $l = n$ (n is the system order).

So-called generic observability is a somewhat less restrictive form of observability in the nonlinear case. A system of the form (7.1) is said to be generically observable if there exists an integer l such that almost any input sequence (generic) of length greater than or equal to l will uniquely determine the state. If strong observability holds, this ensures the existence of an input-output model of the form

$$y(k+1) = F[y(k), y(k-1), \cdots, y(k-n+1), u(k-1), \cdots, u(k-n+1)] \tag{7.11}$$

in a neighbourhood $\Omega \in X \times U$ of the equilibrium state of (7.1). This, in turn, leads to the construction of a state feedback controller for the system whose implementation using neural network based structures is one subject of the next section. Finally, note that a comprehensive treatment of observability of nonlinear systems can, for example, be found in (Isidori 1989).

Some well established results on the control of linear time invariant systems will also be required where only the single-input single-output (SISO) case is considered. In particular suppose that the system of (7.9) (i.e. the pair (A, B)) is controllable. Then this property is equivalent to the existence of a state feedback law $u = Fx$ (where F is a row vector of dimension $1 \times n$ with constant entries) such that the resulting closed loop system is stable, i.e. all eigenvalues of the matrix

$A + BF$ lie inside the open unit circle in the complex plane.

Since

$$x(k+n) = A^n x(k) + [A^{n-1}B, A^{n-2}B, \cdots,]U_n(k) \tag{7.12}$$

the state of the controllable system can be transferred from any initial state to any final state in at most n steps. Also the following can be written

$$y(k+n) = CA^n x(k) + CA^{n-1}Bu(k) + \cdots + CBu(k+n-1) \tag{7.13}$$

Hence if (7.9) is also observable then the following important results are obtained.

1. The state $x(k)$ can be expressed as a linear combination of $y(k), y(k-1), \cdots, y(k-n+1), u(k-1), u(k-2), \cdots, u(k-n)$.

2. The closed loop system can be stabilized using an input which is a linear combination of the signals given in 1 above.

3. If the system has uniform rank d, i.e. $CA^iB = 0$, $0 \leq d-2$ and $CA^{d-1}B \neq 0$, then the input at any time instant k can affect the output only d instants later. The integer d denotes the delay in propagation of the signals through the system and is termed its relative degree.

Suppose now that (7.9) has relative degree d. Then the so-called autoregressive moving average (ARMA) model of its dynamics can be described by either of the following equations (see, for example, (Franklin and Powell 1980))

$$y(k+1) = \sum_{i=0}^{n-1} \alpha_i y(k-i) + \sum_{j=d-1}^{n-1} \beta_i u(k-j) \tag{7.14}$$

$$y(k+d) = \sum_{i=0}^{n-1} \overline{\alpha}_i y(k-i) + \sum_{j=0}^{n-1} \overline{\beta}_j u(k-j) \tag{7.15}$$

In terms of control, it is (7.15) which is the more convenient and the models used

in this chapter for controlling nonlinear systems are based on it.

Let $y_d(k+d)$ denote the desired response at time $k+d$ for the plant under consideration and suppose that its value is known at time k. Then given $x(k)$ it is easy to show that the following choice for $u(k)$ yields the desired output at time $k+d$

$$u(k) = \frac{y_d(k+d) - CA^d x(k)}{CA^{d-1}B} \tag{7.16}$$

This process can be repeated to obtain the desired output d time steps beyond a given instant and the corresponding state equation is

$$x(k+1) = [A - \frac{BCA^d}{CA^{d-1}B}]x(k) + \frac{1}{CA^{d-1}B} y_d(k+d) \tag{7.17}$$

Also if $A - \frac{BCA^d}{CA^{d-1}B}$ has all its eigenvalues inside the open unit circle in the complex plane (i.e. stable) this tracking can be obtained with bounded inputs and the system is termed minimum phase.

In qualitative terms, minimum phase implies that a bounded output guarantees a bounded input. If a linear system is not minimum phase, asymptotic tracking is not possible. Alternatively, the minimum phase property can be interpreted as requiring that all zeros of the system transfer function lie in the open unit circle in the complex plane.

7.3 Neural Network Based Control of Nonlinear Systems

In terms of the control of plants described by (7.1), the approach used will depend on the information available about the system and the control objectives. Two possible scenarios in terms of the information available about the plant are

1. f and h are known.

2. f and h are unknown or, at best, partially known.

The first case arises when the laws governing the system are known but the states of the system are not accessible (or cannot be physically measured) or are only partially accessible, eg a mechanical system where the velocities cannot be measured directly. By definition, knowledge of the states will enable accurate prediction of the system response and hence the observation problem here is actually that of estimating the state based on input and output observations over a time interval, say, $[k_0, k_0 + l]$. This is the well known observer problem.

The second case arises most often when dealing with complex systems for which first principles based laws are not available. The essential task now is to create a model whose input-output behaviour 'closely approximates' (ideally is identical to) that of the system over the range of operation (of the system). In this case, identification must be performed using the system itself. Also since there is no prior knowledge available, this must be undertaken in an open loop fashion and this, in turn, requires one of the following two conditions to hold.

1. Bounded outputs - if the inputs are in a bounded set, the resulting system outputs will also belong to a bounded set (with an appropriate definition of a bounded set).

2. Ability to reset the system - this is an alternative assumption which is not considered in this work.

It is now necessary to make precise the tracking and regulation control problems discussed briefly in the previous section. In terms of (7.1) these problems are defined as follows.

1. State Regulation (Stabilization) - using only input-output data determine a control law that will stabilise the overall system around a pre-specified equilib-

rium point.

2. Tracking - given $y_d(k)$ as a uniformly bounded desired output sequence, determine an input $u(k)$ such that $\lim_{k\to\infty} ||y(k) - y_d(k)|| = 0$. The so-called output regulation problem arises when convergence to a pre-specified fixed value (assumed to be zero) is required.

Clearly the regulation problem only has a non-trivial solution when the system is unstable. This problem is (relatively) easier if all the states are accessible so here the interest will be on first estimating the state via an observer and then using the resulting state estimate to stabilize the system. This problem is considered next where f and h are assumed known.

7.3.1 State Reconstruction

It is first necessary to derive conditions for local observability of the nonlinear system (7.1), i.e. given the origin as an equilibrium state, does there exist a region, say Ω_x around the origin such that any state $x \in \Omega_x$ can be uniquely determined by probing the system with any input sequence of sufficient length. Equivalently, conditions are sought under which the system is locally strongly observable. The following result (Levin and Narendra 1993) gives sufficient conditions for strong local observability of (7.1) in terms of its linearisation at the origin

$$\begin{aligned} \delta x(k+1) &= f_x|_{0,0}\delta x(k) + f_u|_{0,0}\delta u(k) = A\delta x(k) + B\delta u(k) \\ \delta y(k) &= h_x|_0\delta x(k) = C\delta x(k) \end{aligned} \tag{7.18}$$

where $A := f_x|_{0,0}$, $B := f_u|_{0,0}$, and $C := h_x|_0$ are the system's Jacobian matrices.

Theorem 4 *If the linear system (7.18) is observable then the nonlinear system (7.1) is locally strongly observable.*

7.4 Conclusions

This chapter has presented some essential theory for the control of nonlinear systems which underlies the neural network training processes described in the next chapter. It is now possible to describe the plant to be controlled as locally strongly observable and, with the aid of the Lyapunov and contraction mapping theories, as stabilizable. The important concepts of identification and controllability are closely related to observability and stabilizability respectively. These properties are directly derivable from the linearisation of the nonlinear system around the equilibrium (which is assumed here to be the origin). The theory will now be used in the implementation of neural network schemes for control in the next chapter. The examples to be presented are restricted to Single Input Single Output (SISO) plants, but since neural networks are easily extended to Multiple Input Multiple Output (MIMO) schemes, a MIMO notation is used.

Chapter 8

Implementation of Neural Network Based Nonlinear Control

8.1 Introduction

This chapter considers the implementation of feedback control schemes based on neural networks. The key tasks of state estimation, system identification, and stabilisation/tracking (sections 8.2.1 to 8.2.3) are each considered in turn with training procedures that build on the theoretical concepts described in the previous chapter. Following this, the use of these components in a multiple model adaptive control scheme (the linear model case was presented in chapter 6) is treated (section 8.3). Section 8.4 describes the parallelisation of the scheme with section 8.5 presenting the results. The chapter ends on some performance analysis (section 8.6).

8.2 Neural Network Based Control of Nonlinear Systems

Before a switching scheme can be constucted for the control of nonlinear plants, the neural network components of that scheme must first be implemented. This section describes the training processes (with results) of neural networks for state reconstruction, identification, stabilisation and tracking.

8.2.1 State Reconstruction

In section 7.3.1 the concept of strong local observability was introduced. This concept is essential to the type of plant which is to be controlled in this chapter. Provided the plant is observable within the region of interest around the equilibrium point, then a neural network can be trained to estimate the states.

Network Implementation: Since the system is assumed to be strongly observable in its range of operation, then there exists a mapping $\overline{\Phi}$ such that $x(k) = \overline{\Phi}[Y_n(k), U_{n-1}(k)]$, where here it is assumed that the system order n is known. (If only an upper bound, say $\overline{n}$, of the system order is known then all algorithms must be suitably modified to work with $\overline{n}$ instead of n.) For control purposes it is essential to study the state of the system after the probing input has been applied. Since $x(k+n-1) = F_{n-1}[x(k), U_{n-1}(k)]$, there exists a function Φ such that

$$x(k+n-1) = \Phi[Y_n(k), U_{n-1}(k)] \tag{8.1}$$

or, on rearranging the indexes,

$$x(k) = \Phi[Y_n(k+n-1), U_{n-1}(k-n+1)] \tag{8.2}$$

Given f and h, the variables on both sides of (8.1) can be observed and hence

a feedforward neural network, denoted NN_Φ, can be trained to emulate Φ and (by assumption at this stage f and h are known) the training procedure can be completed off-line. At each k, the inputs to the network (not to be confused with the inputs to the system) are the past $n-1$ inputs and the past n outputs (a total of $2n-1$). The output is the estimated state $\hat{x}(k)$ which is then compared with the state of a simulated system and the error is given by

$$e_x(k) = x(k) - NN_\Phi[Y_n(k-n+1), U_{n-1}(k-n+1)] \tag{8.3}$$

The training procedure for the observer requires the adjustment of the parameters of NN_Φ along the negative gradient of $||e_x(k)||$. Suppose now that θ denotes a parameter of NN_Φ and η the learning rate. Then the update rule is given by

$$\theta(k+1) = \theta(k) + \eta e_x(k) \frac{\partial \hat{x}(k)}{\partial \theta}|_{\theta=\theta(k)} \tag{8.4}$$

Figure 8.1 is a schematic of the observer learning process where TDL denotes a tapped delay line. Next the details of this process are discussed and some performance enhancing actions are developed.

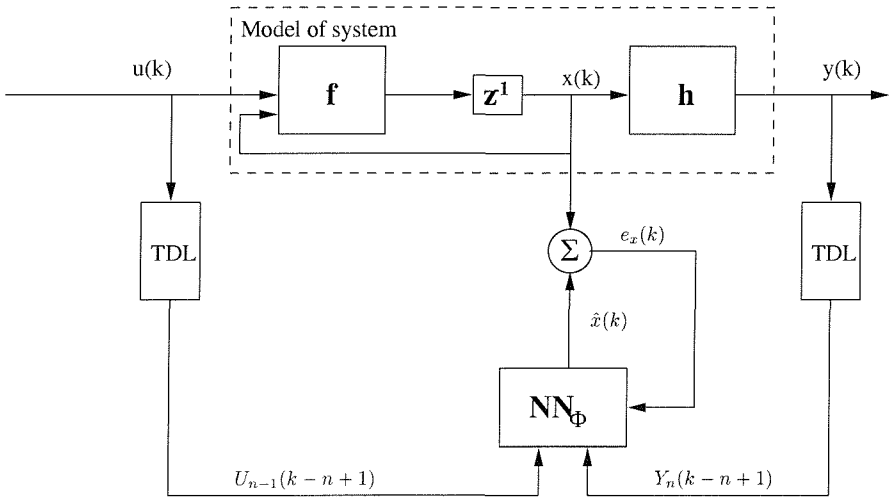


Figure 8.1: Training an observer (Levin and Narendra 1996)

The specific error function to be minimised is:

$$\begin{aligned} I &= \frac{1}{2}\sum_j \left(x_j(k) - NN_\Phi[Y_n(k-n+1), U_{n-1}(k-n+1)]\right)^2 \\ &= \frac{1}{2}\sum_j \left(x_j(k) - \hat{x}_j)\right)^2 \end{aligned} \tag{8.5}$$

and the weights are updated using (with $x_j(k) = y_j^{(L)}$):

$$\begin{aligned} \Delta w_{ij}^{(L-1)} &= \eta(x_j(k) - \hat{x}_j(k))\frac{\delta \hat{x}_j(k)}{\delta w_{ij}^{(L-1)}} \\ &= \eta e_x(k)\hat{x}_j(k)(1-\hat{x}_j(k)) \end{aligned} \tag{8.6}$$

The neural network is a Multi-Layer Perceptron (MLP) and the notation adopted is that defined in section 2.3, i.e.

$$NN^L_{n_0,n_1,n_2,\cdots,n_L} \tag{8.7}$$

where NN is an L-layer network with n_l nodes at the l^{th}-layer, n_0 denotes the input layer and n_L the output layer. It should also be noted that the number n_l does not include the bias node θ which is present in all layers except L.

Results

The neural network was initially trained on an example in (Levin and Narendra 1996). The third-order system employed is described by

$$\begin{aligned} x_1(k+1) &= 1.4x_2(k) - 0.5x_3(k) + 0.3u^2(k) \\ x_2(k+1) &= x_1(k) + [1 - 0.3x_2(k)]u(k) \\ x_3(k+1) &= 0.4x_1(k)x_2(k) - x_3(k) + u(k) \\ y(k) &= x_1(k) \end{aligned} \tag{8.8}$$

A fixed training parameter, $\eta = 0.55$, was used with a momentum value $\alpha = 0.95$. Weights were initialised to random values in the range (-0.1, 0.1). Scaling factors were placed on the sigmoid output nodes with values: $\hat{x}_1 \in (-8, 8)$, $\hat{x}_2 \in (-8, 8)$, $\hat{x}_3 \in (-8, 8)$. This means that the upper bound of the sigmoid function of 1 is interpreted as the upper bound of the x-range (8 in these cases) and the lower bound of 0 as the lower bound of x (-8) and $NN_\Phi \subset NN^3_{5,10,5,3}$. The results are shown in figure 8.2 and are further discussed next together with some necessary alterations.

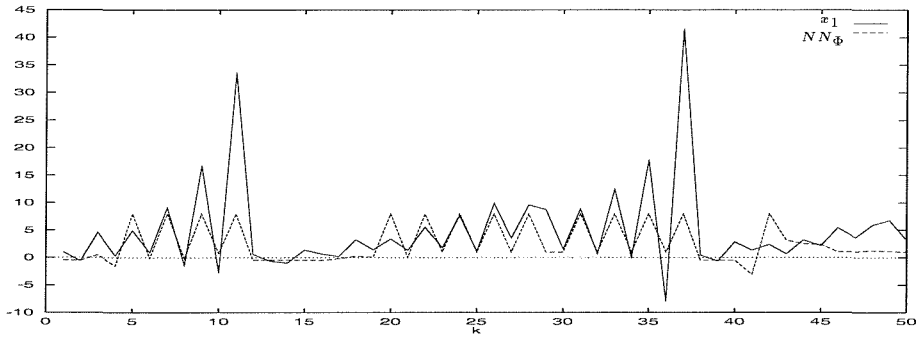
Improvements and Alterations

One immediate alteration that was necessary in order to train the network at all, in any practical sense, was to set a boundary (a range of values for the plant output outside of which the plant states are reset to zero) for the output of the plant model (equations (8.8)). In the results in figure 8.2 this is set at $y \in (-20, 20)$.

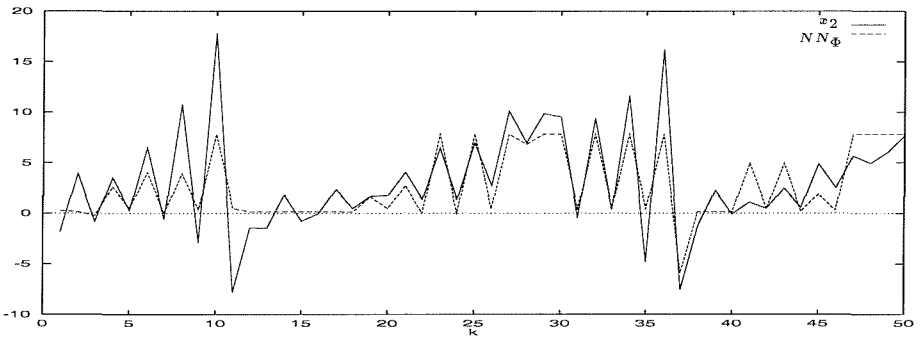
As can be seen from figure 8.2, the estimation (or closeness of tracking) is accurate only within the x-boundaries. This is probably due to the large changes in the weights that will result when the plant is outside these boundaries. The estimation is particularly poor around the origin.

In an effort to improve estimation around the origin, a training bias was introduced. This simply involved training the network on small deviations from the origin for a number of steps (denoted by o) each time the system is reset.

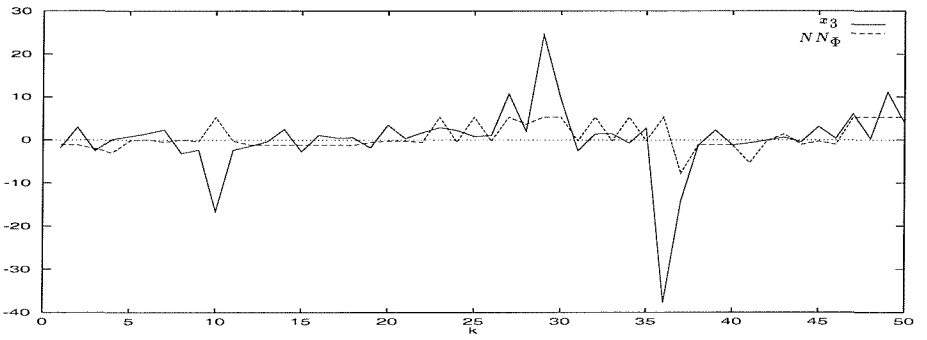
An example of origin training with $o = 25$ and a boundary $y \in (-20, 20)$ is shown in figure 8.3. Random state and control input deviations were chosen to be in the following ranges for all training examples involving an origin bias: $x_o \in (-0.05, 0.05)$, $u_o \in (-0.01, 0.01)$ (only during the bias training stages). It



(a) $\hat{x}_1$



(b) $\hat{x}_2$



(c) $\hat{x}_3$

Figure 8.2: Estimation of states with a boundary set at $y \in (-20, 20)$.

can be seen that, although accuracy around the origin is improved slightly, this is largely at the expense of accuracy elsewhere - particularly, in this case, in the estimation of x_3 (figure 8.3c). For easier inspection, figure 8.4 shows a plot of the errors in estimation in figures 8.2 and 8.3.

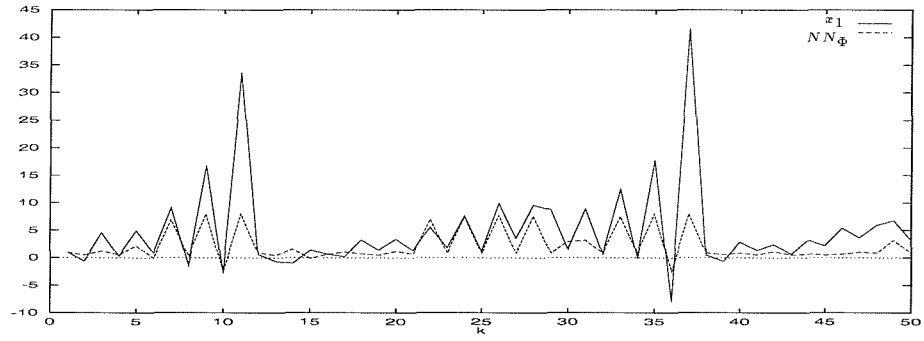
Two comments can now be made on these initial results. Firstly, despite a boundary reset of 20, it has been possible for the plant to reach heights of more than twice this value before being reset. This would advocate a narrowing of the boundary. Secondly, the origin training has too strong a bias, any improved accuracy around the origin is outweighed by the decrease in accuracy elsewhere.

Figure 8.5 shows the case when the boundary is reduced to $y \in (-5, 5)$ without an origin training bias. Figure 8.6 shows the boundary $y \in (-5, 5)$ with a training bias $o = 25$. Figure 8.7 shows the error plots.

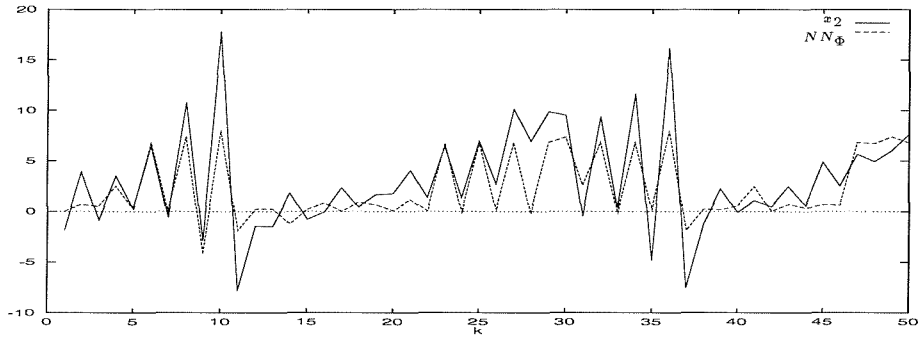
Here, the benefit of an origin training bias has begun to appear. Although the estimation of x_1 is about as good with or without the bias, the improvement in estimation of x_2 and x_3 is marked. If the stabilisation stage is taken into account, it can be seen that the operation boundary will be such that the norm of the state vector is assumed to be $S = \{x | ||x|| \leq 2\}$ which in this example means a narrowing of the boundary can still be afforded.

Figure 8.8 shows the case when the boundary is reduced to $y \in (-3.5, 3.5)$ without an origin training bias. Figure 8.9 shows the boundary $y \in (-3.5, 3.5)$ with a training bias $o = 25$. Figure 8.10 shows the error plots.

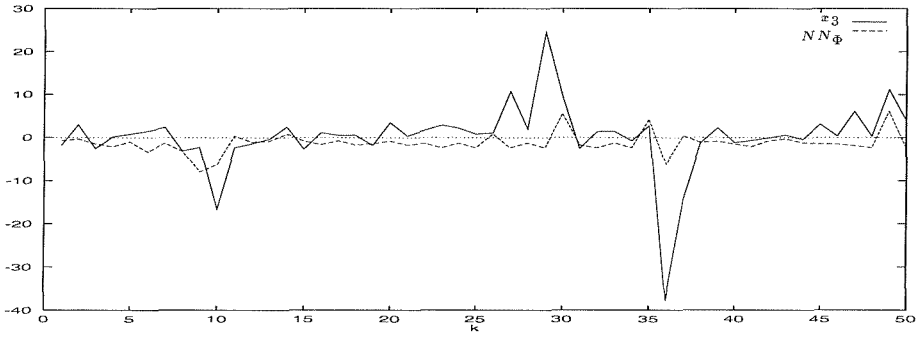
In these graphs, it can be seen that the effectiveness of the origin bias is negligible (i.e. that the errors are roughly the same for the estimation with and without an origin training bias). The ineffectiveness of this bias is probably due to the size of the boundary involved. In any case, training is now concentrated



(a) $\hat{x}_1$

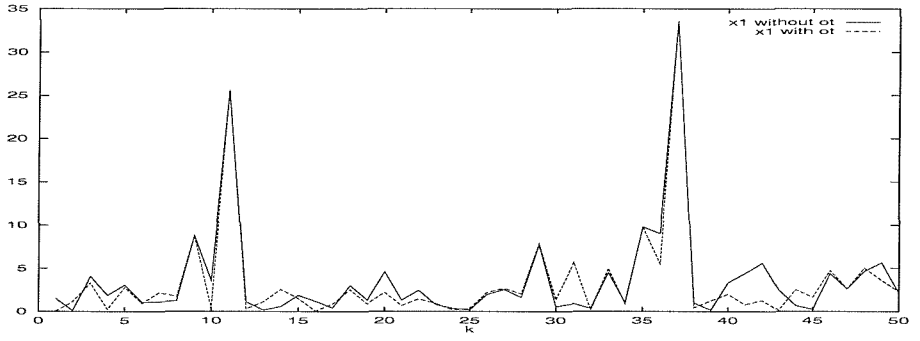


(b) $\hat{x}_2$

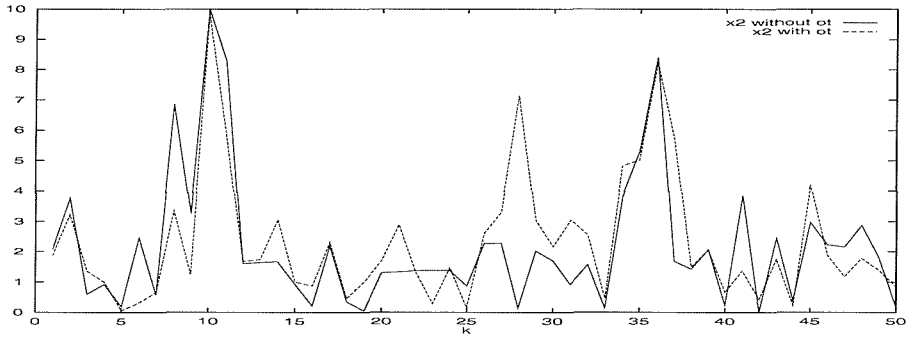


(c) $\hat{x}_3$

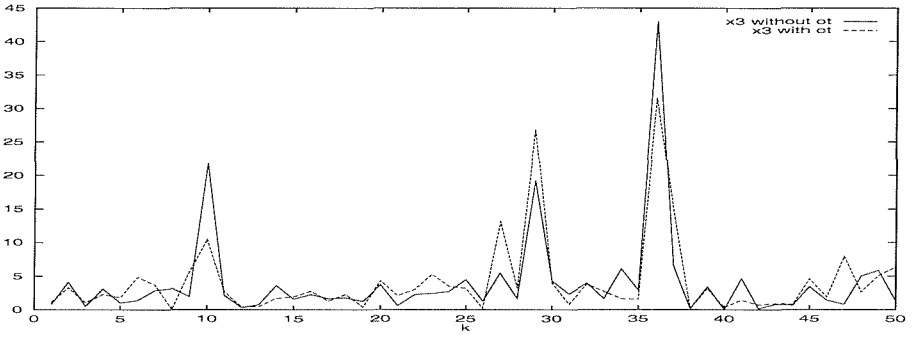
Figure 8.3: Estimation of states with a boundary set at $y \in (-20, 20)$ and with origin training bias of $o = 25$.



(a) $\hat{x}_1$

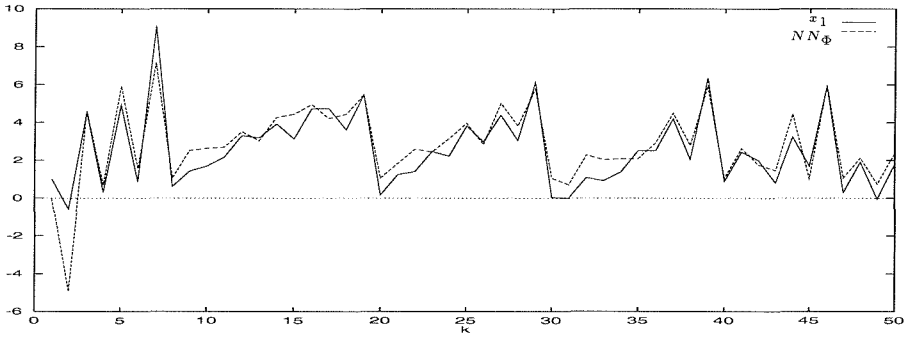


(b) $\hat{x}_2$

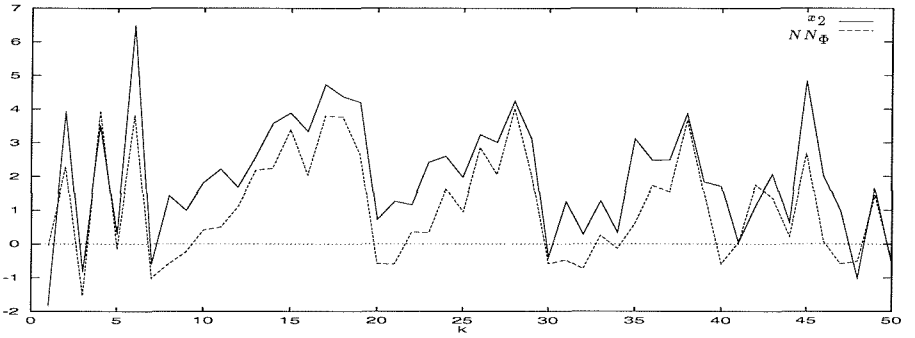


(c) $\hat{x}_3$

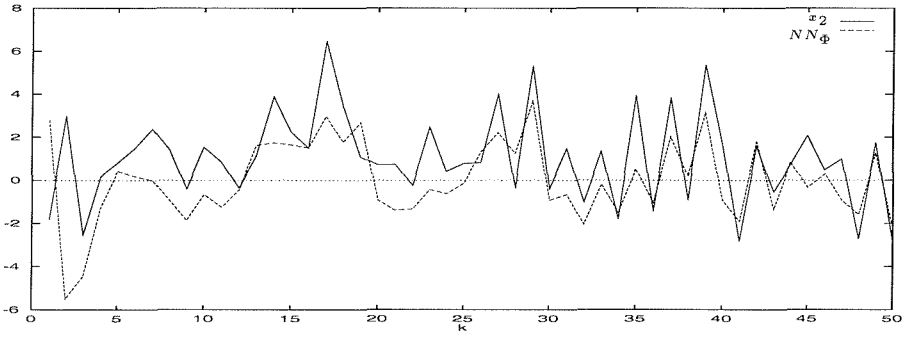
Figure 8.4: Comparison of error with boundary $y \in (-20, 20)$



(a) $\hat{x}_1$

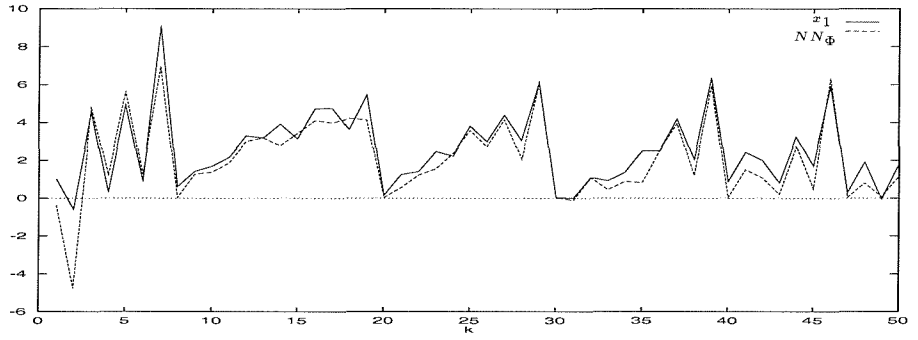


(b) $\hat{x}_2$

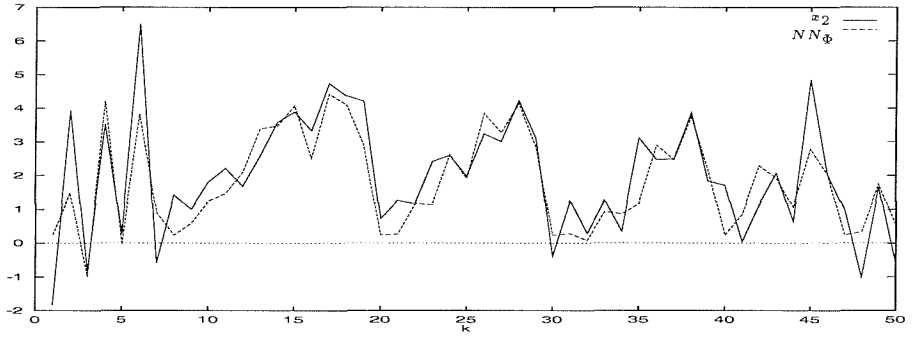


(c) $\hat{x}_3$

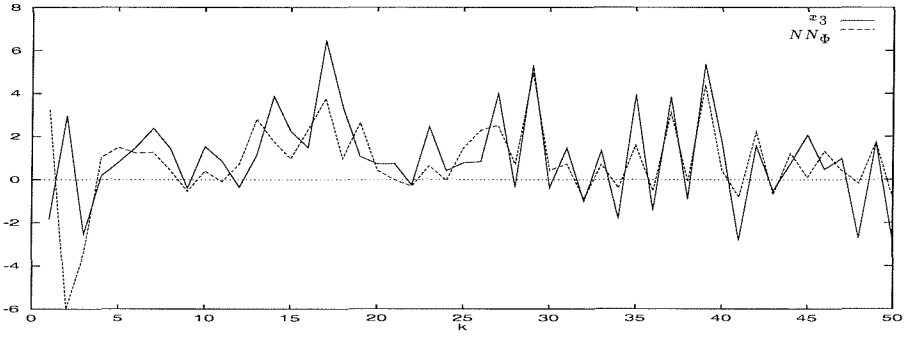
Figure 8.5: Estimation of states with a boundary set at $y \in (-5, 5)$.



(a) $\hat{x}_1$

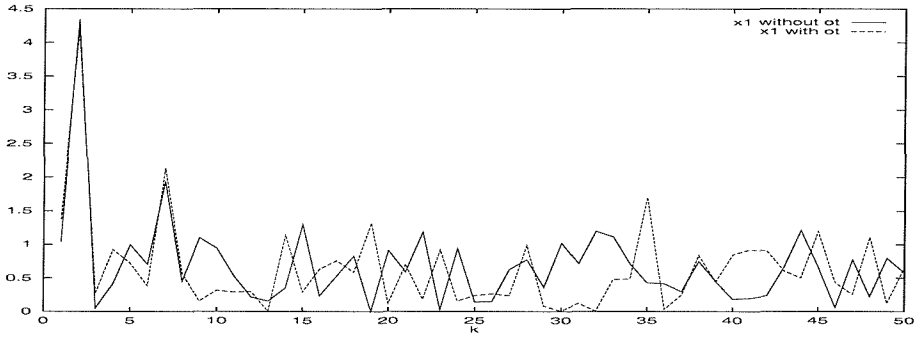


(b) $\hat{x}_2$

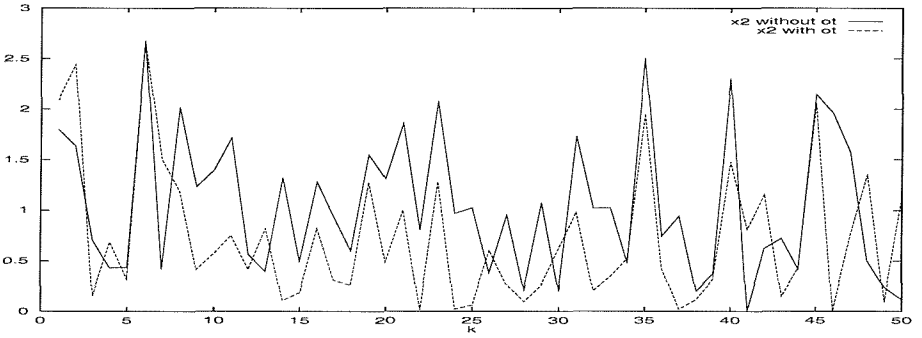


(c) $\hat{x}_3$

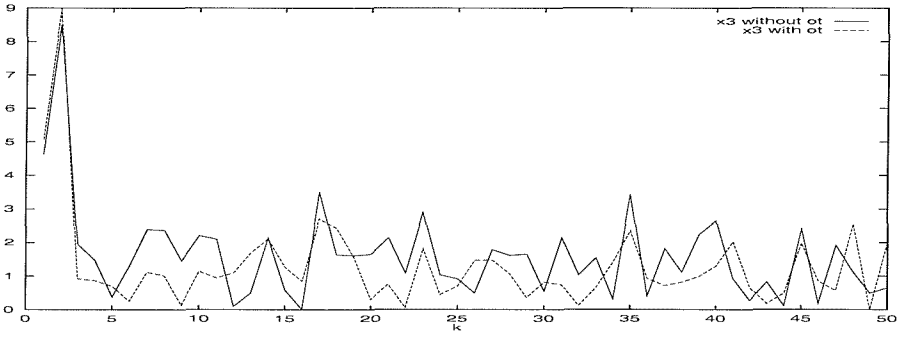
Figure 8.6: Estimation of states with a boundary set at $y \in (-5, 5)$ and with origin training bias of $o = 25$.



(a) $\hat{x}_1$

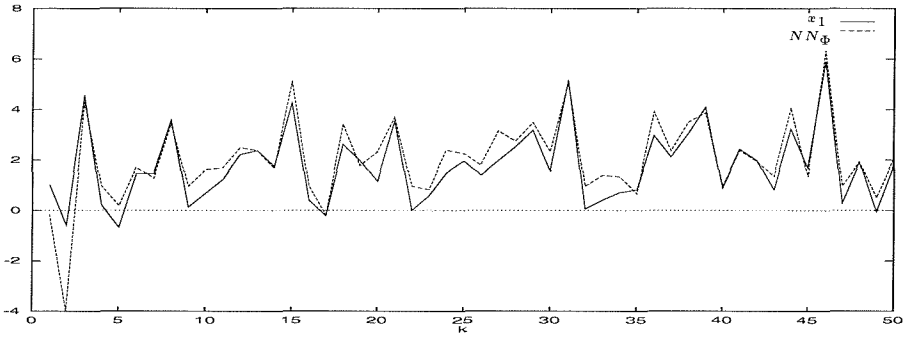


(b) $\hat{x}_2$

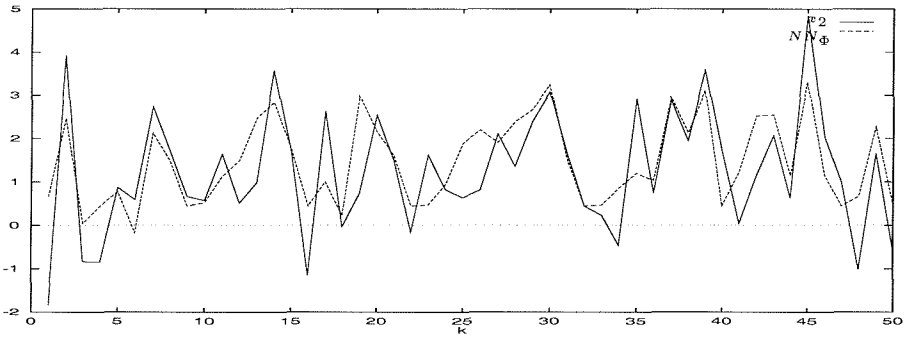


(c) $\hat{x}_3$

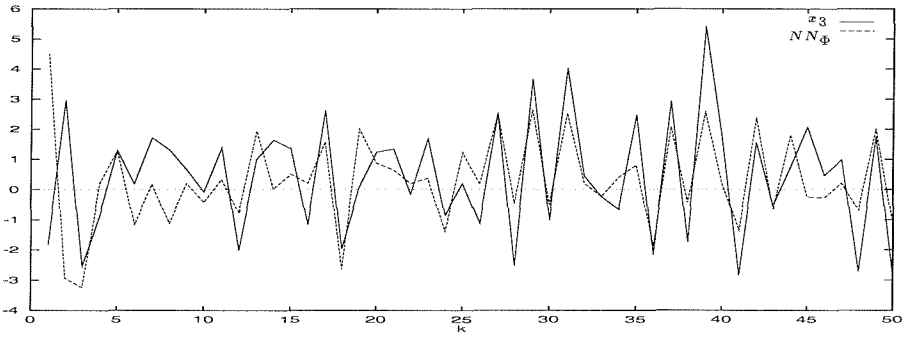
Figure 8.7: Comparison of error with boundary $y \in (-5, 5)$



(a) $\hat{x}_1$

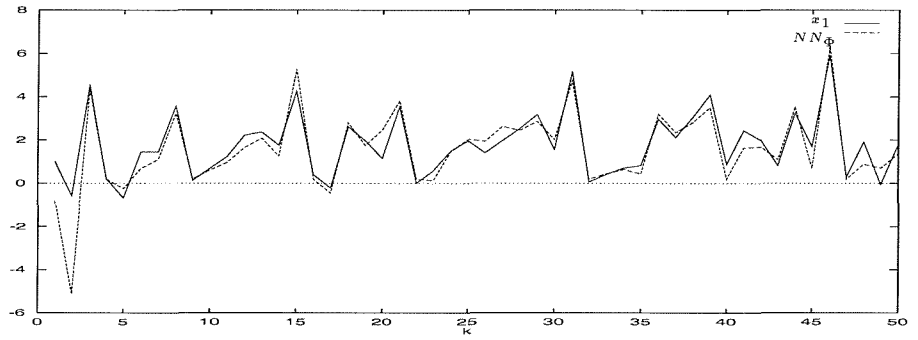


(b) $\hat{x}_2$

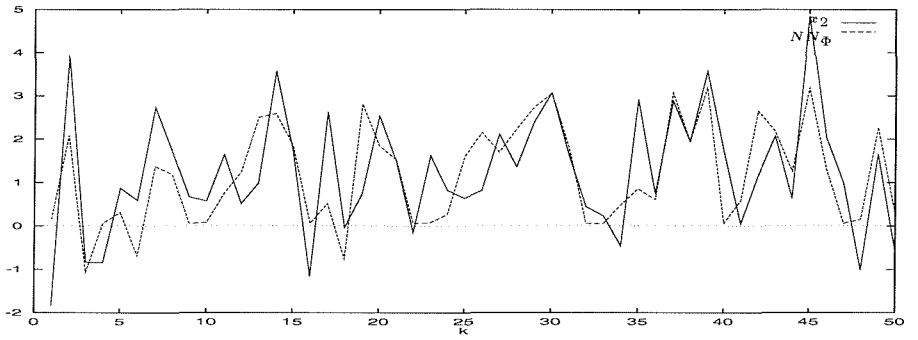


(c) $\hat{x}_3$

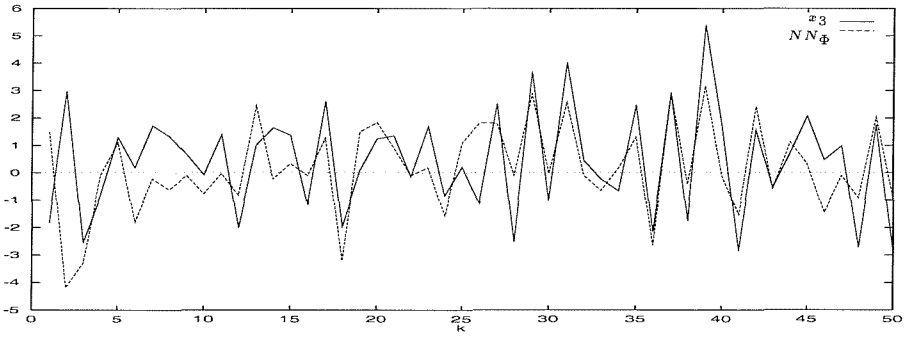
Figure 8.8: Estimation of states with a boundary set at $y \in (-3.5, 3.5)$.



(a) $\hat{x}_1$

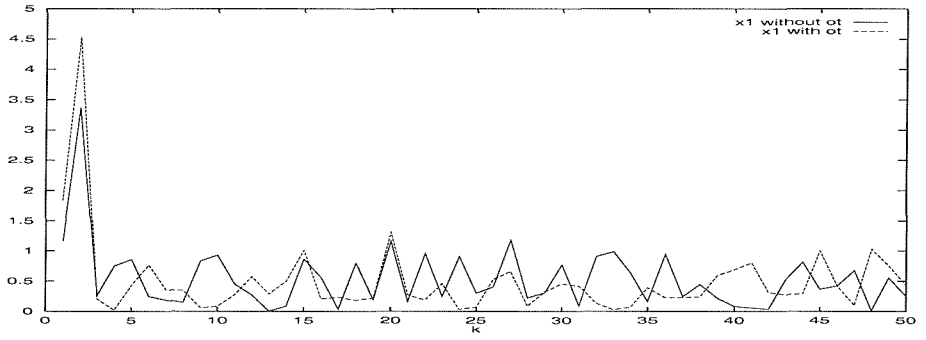


(b) $\hat{x}_2$

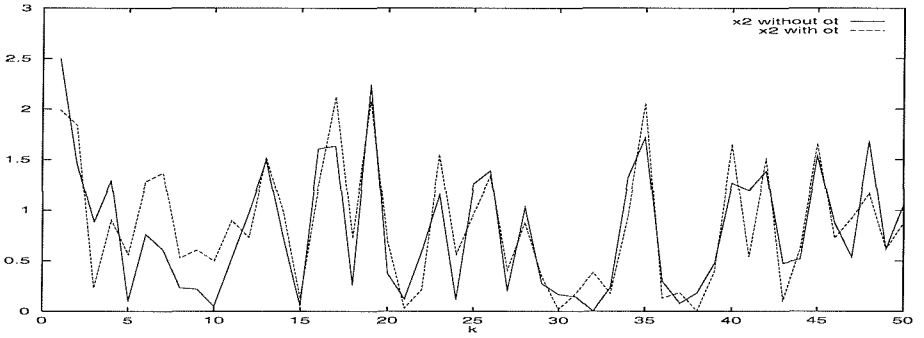


(c) $\hat{x}_3$

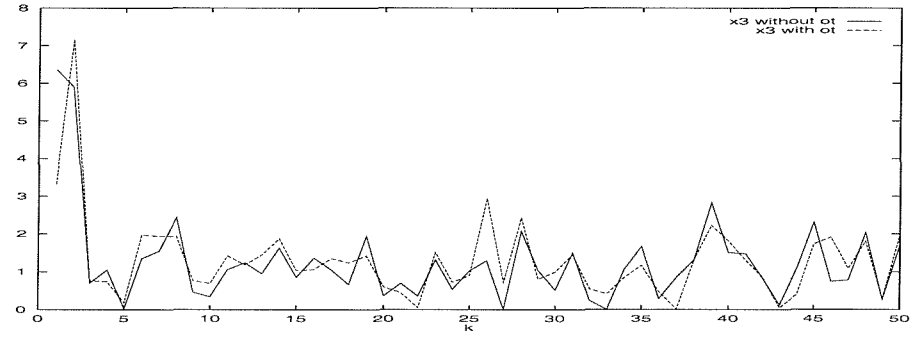
Figure 8.9: Estimation of states with a boundary set at $y \in (-3.5, 3.5)$ and with origin training bias of $o = 25$.



(a) $\hat{x}_1$



(b) $\hat{x}_2$



(c) $\hat{x}_3$

Figure 8.10: Comparison of error with boundary $y \in (-3.5, 3.5)$

around the origin, regardless of biasing - this may in turn point to a possibility of increasing the random deviations involved in the origin training process.

These last modifications do not greatly reduce the errors relative to the previous boundary of $y \in (-5, 5)$. Figure 8.11 compares the error in estimation of the boundaries $y \in (-3.5, 3.5)$ and $y \in (-5, 5)$. They do not vary greatly, however, due to the system being reset more frequently in the narrower boundary, these two cases can not be compared exactly.

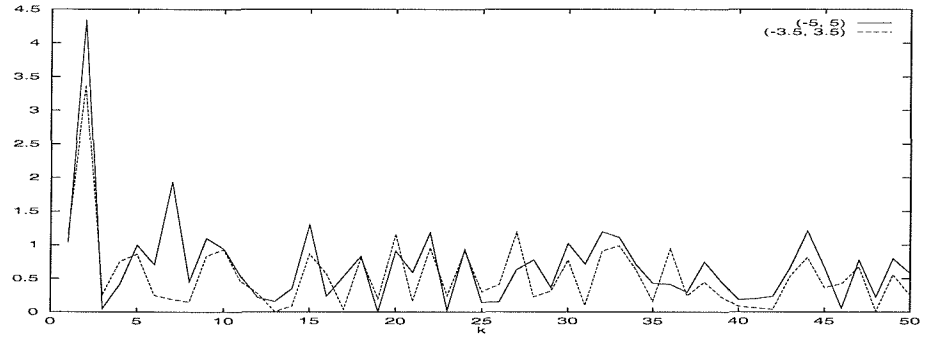
Finally, it should be noted that the accuracy of estimation can be increased greatly by training an observer off raw state inputs. This is equivalent to training a network $NN_f \in NN^3_{4,10,5,3}$ to map the function f in figure 8.1. Figures 8.12 to 8.15 show equivalent plots for some of the cases discussed so far.

Despite the increased accuracy, this situation is unlikely to be acceptable, if the raw states are available during run time, then a better solution might be to train the stabiliser network straight off them. Of course, that is only possible from a model that can be used to calculate the next state from the current states $\mathbf{x}$ and input u.

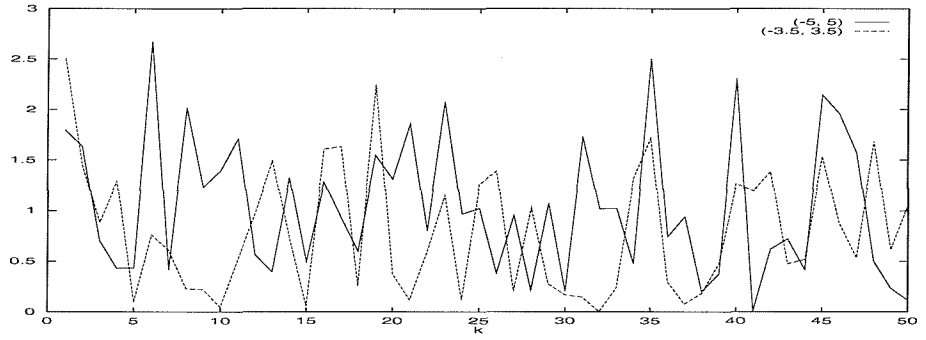
8.2.2 Identification

This is a very extensively studied area for the case of linear dynamics - see, for example, (Ljung 1999). In this case, if the system order is known then the structure of the model can be chosen and the remaining task is parameter estimation, eg the parameters $\overline{\alpha_i}$ and $\overline{\beta_i}$ of (7.15). This does not apply in the nonlinear case where the structure of the model has to be justified.

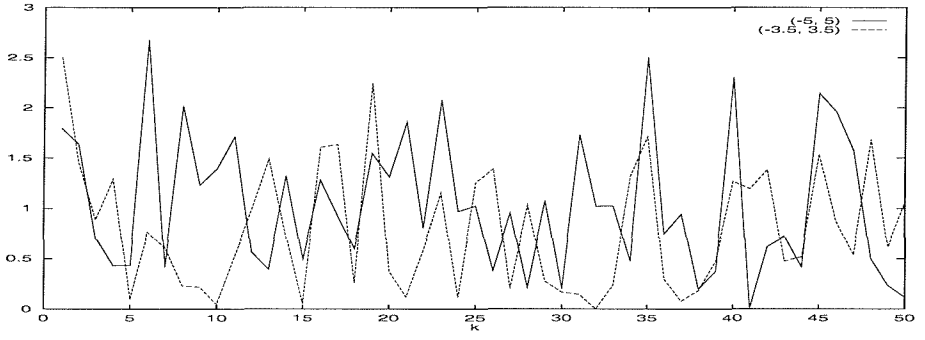
The true system is not known at this juncture and hence it must be assumed that it belongs to a specified set. This then leads to the assumption that a pa-



(a) $\hat{x}_1$

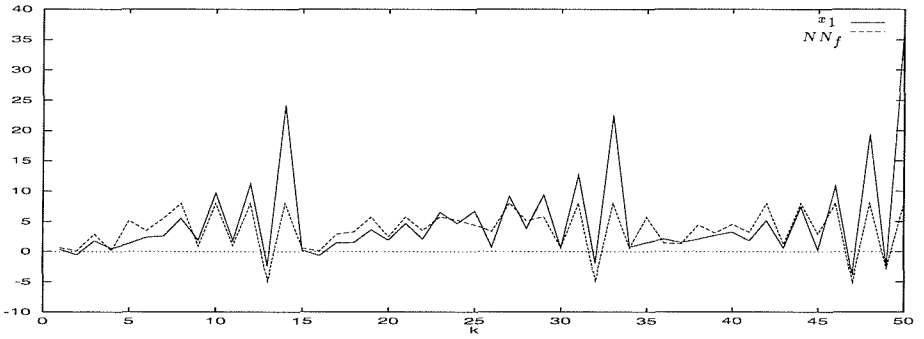


(b) $\hat{x}_2$

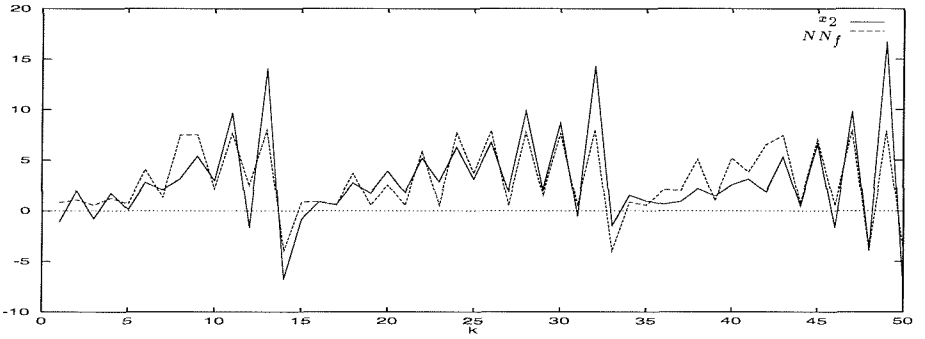


(c) $\hat{x}_3$

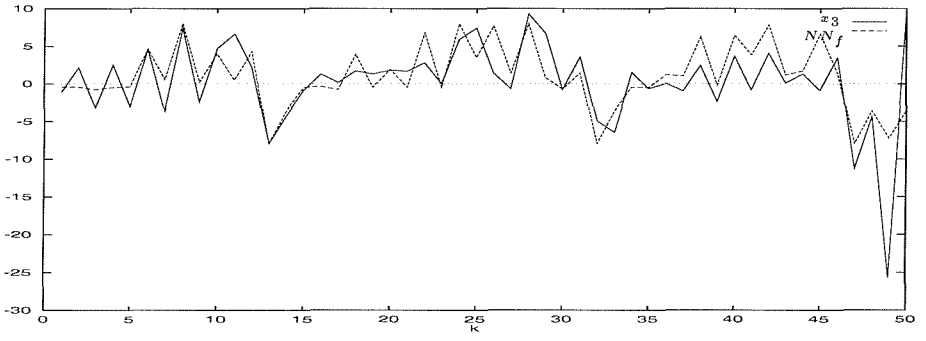
Figure 8.11: Comparison of error with boundary $y \in (-3.5, 3.5)$ and $y \in (-5, 5)$



(a) $\hat{x}_1$

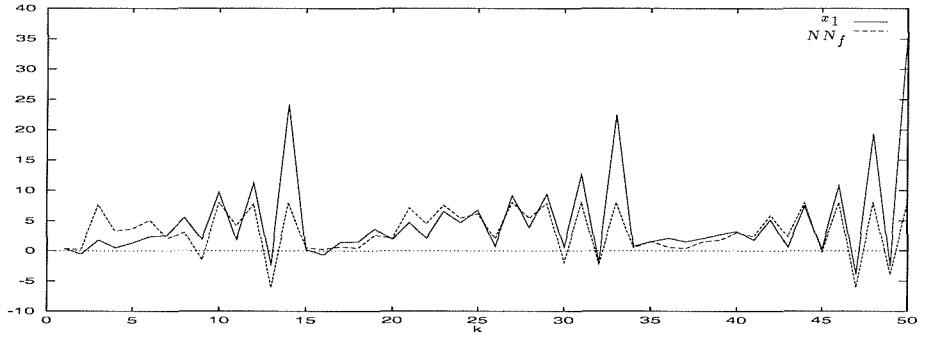


(b) $\hat{x}_2$

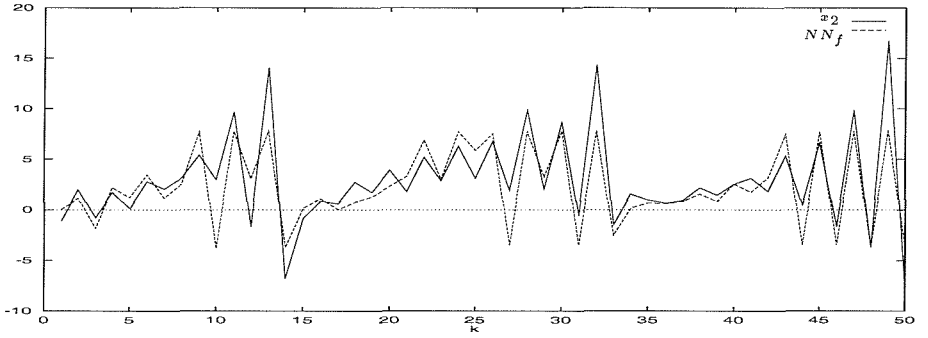


(c) $\hat{x}_3$

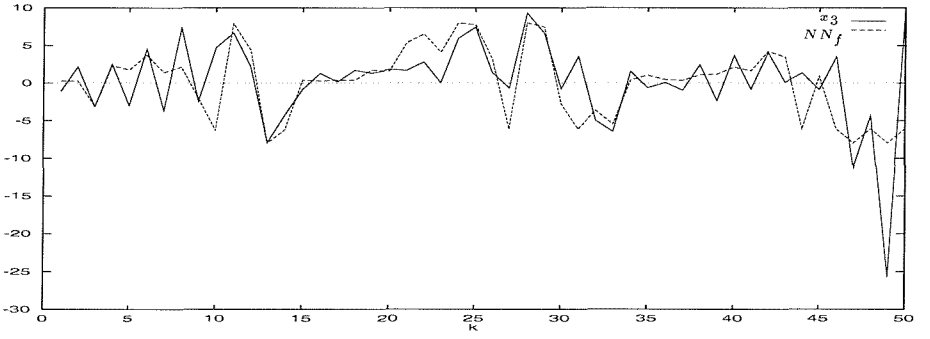
Figure 8.12: Estimation of states with a boundary set at $y \in (-20, 20)$.



(a) $\hat{x}_1$

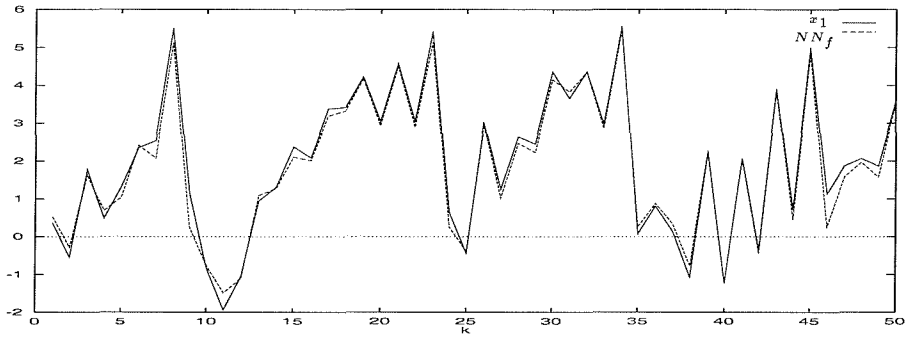


(b) $\hat{x}_2$

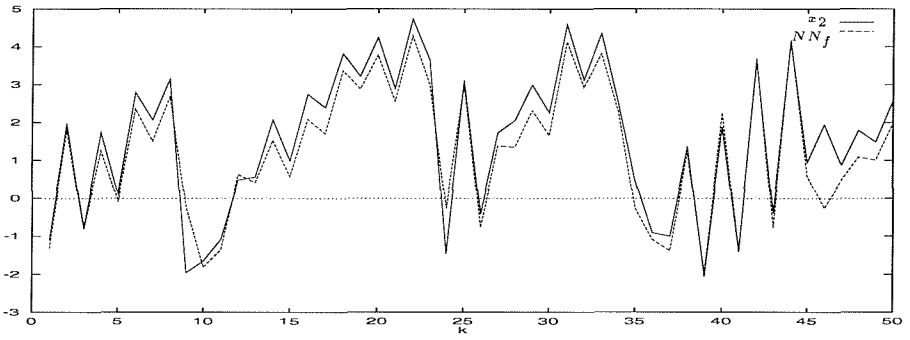


(c) $\hat{x}_3$

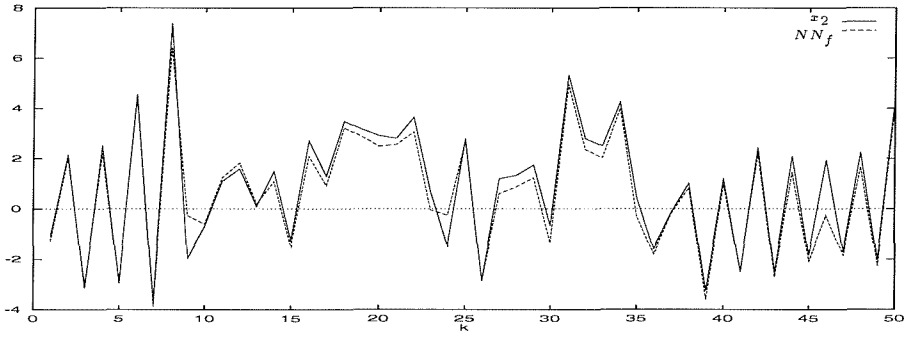
Figure 8.13: Estimation of states with a boundary set at $y \in (-20, 20)$ and with origin training bias of $o = 25$.



(a) $\hat{x}_1$

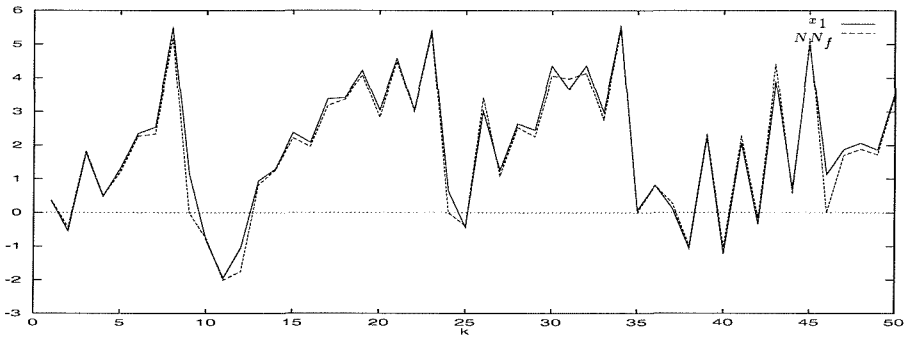


(b) $\hat{x}_2$

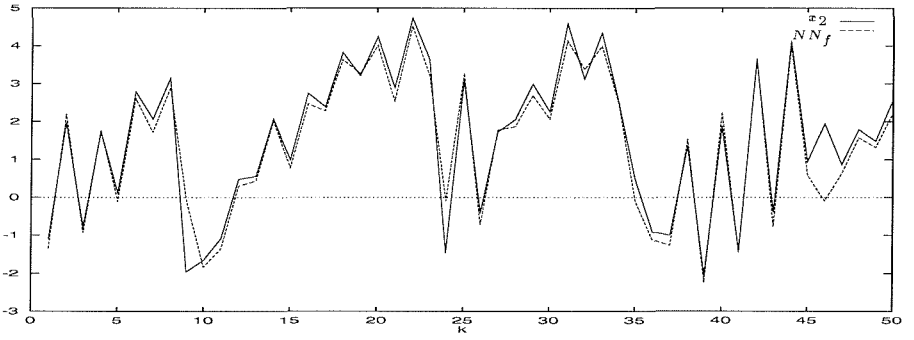


(c) $\hat{x}_3$

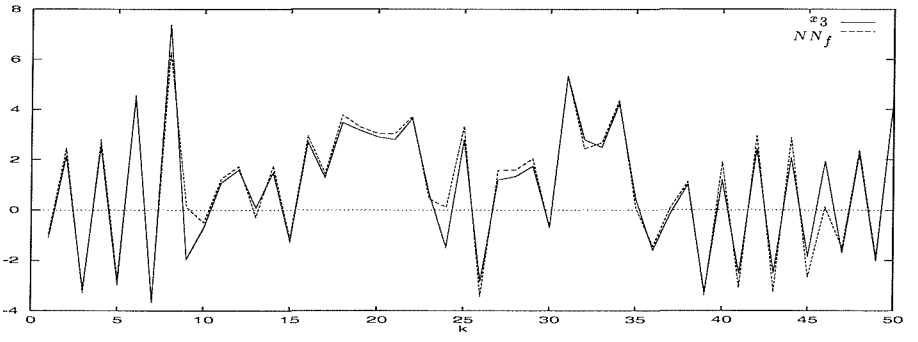
Figure 8.14: Estimation of states with a boundary set at $y \in (-5, 5)$.



(a) $\hat{x}_1$



(b) $\hat{x}_2$



(c) $\hat{x}_3$

Figure 8.15: Estimation of states with a boundary set at $y \in (-5, 5)$ and with origin training bias of $o = 25$.

rameterised model can be chosen that (theoretically) can realize the input-output behaviour of any member of that set. Hence in this setting identification again reduces to a parameter estimation problem.

Given the approximation capabilities of feedforward neural networks, the functions f and h can be approximated by a multilayed neural network with appropriate input and output dimensions. Hence (7.1) can be realised by a system of the form

$$\begin{aligned} z(k+1) &= NN_f[z(k), u(k)] \\ \hat{y}(k) &= NN_h[z(k)] \end{aligned} \tag{8.9}$$

Here the system's states are assumed not to be accessible and hence the training of such a network to identify the system requires the use of dynamic backpropagation - a computationally very intensive procedure which is hence hard and slow to implement.

Suppose now (as in the linear case) that it is possible to determine the future outputs of the system as a function of the past observations of the inputs and outputs. Equivalently, there exists a number l and a continuous function $\overline{h}$: $Y_l \times U_l \rightarrow Y$ such that the recursive (input-output) model

$$y(k+1) = \overline{h}[y(k), y(k-1), \cdots, y(k-l+1), u(k), u(k-1), \cdots, u(k-l+1)] \tag{8.10}$$

has the same input-output behaviour as the original system (7.1). Then $\overline{h}(\cdot)$ can

be realised by a feedforward neural network. This results in the model

$$y(k+1) = NN_{\bar{h}}[y(k), y(k-1), \cdots, y(k-l+1), u(k), u(k-1), \cdots, u(k-l+1)] \tag{8.11}$$

Since the outputs and inputs of the network are directly accessable at all time instants, backpropagation (or indeed any other supervised training method) can be used to train the network. Also if the conditions for local observability hold then it can be shown (Levin and Narendra 1993) that locally the system can be described by an input-output model.

Network Implementation: If the strong observability assumptions hold in the system's region of operation, then the identification procedure using a feedforward neural network is a straightforward task. At each k, the past n inputs and the past n outputs are fed into the network - figure 8.16. The networks's output is then compared with the next observation of the system's output, to yield a prediction error

$$e(k+1) = y(k+1) - NN_{\bar{h}}(Y_n(k-n+1), U_n(k-n+1) \tag{8.12}$$

The network weights are then adjusted using backpropagation to minimise the least squares error. The specific error function to minimise is:

$$\begin{aligned} E &= \frac{1}{2}\sum_j (y_j(k+d) - NN_d[Y_n(k-n+1), U_{n-1}(k-n+1)])^2 \\ &= \frac{1}{2}\sum_j (y_j(k+d) - \hat{y}_j(k+d))^2 \end{aligned} \tag{8.13}$$

and are updated using (with $\hat{y}_j(k+d) = y_j^{(L)}$):

$$\Delta w_{ij}^{(L-1)} = \eta(y_j(k+d) - \hat{y}_j(k+d))\hat{y}_j(k+d)(1-\hat{y}_j(k+d))y_i^{(L-1)} \qquad (8.14)$$

Figure 8.16 includes a stabilisation loop for examples where the nonlinear plant requires stabilisation first. If, however, the plant is open loop stable then the model M_s can be substituted for the model M. For examples where the plant requires stabilisation $U_{(s)n-1}(k-n+1)$ is defined as:

$$U_{(s)n-1}(k-n+1) = [u(k-n+1) + u_s(k-n+1), u(k-n+2) + u_s(k-n+2), \cdots, u(k-1) + u_s(k-1)] \qquad (8.15)$$

Stabilization with neural networks is treated in the next section.

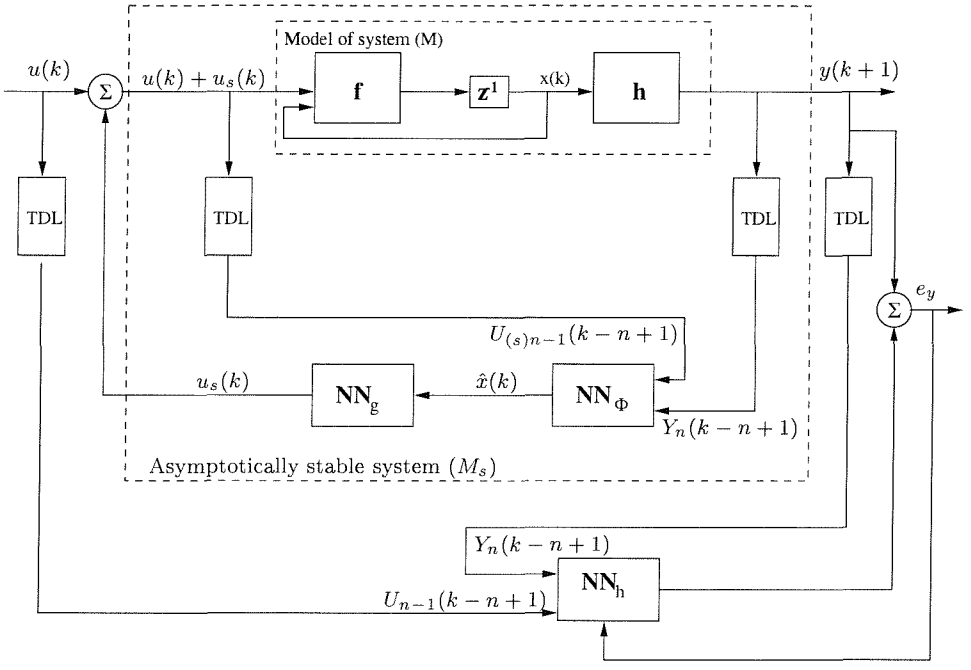


Figure 8.16: The first stage of training an identifier

Once the identification is complete, two modes of operation are possible as discussed next.

Series-Parallel Mode: - the outputs of the actual system are used as inputs to the model. Hence this scheme can only be used in conjunction with the system and can generate only one step ahead prediction. The architecture is identical to that used for identification.

Parallel Mode: - this must be used for more than one step ahead prediction. Here the output of the network is fed back into the network - figure 8.17 - and hence the outputs of the network itself are used to generate future predictions (i.e. a recurrent network). If the relative degree of the plant is d then the output at time $k + d$ is a function of the state and the input at time k only. Since it is independent of inputs introduced after time k, these can be arbitrarily set to zero (Levin and Narendra 1996). The accuracy of such predictions is only realistic for short horizons.

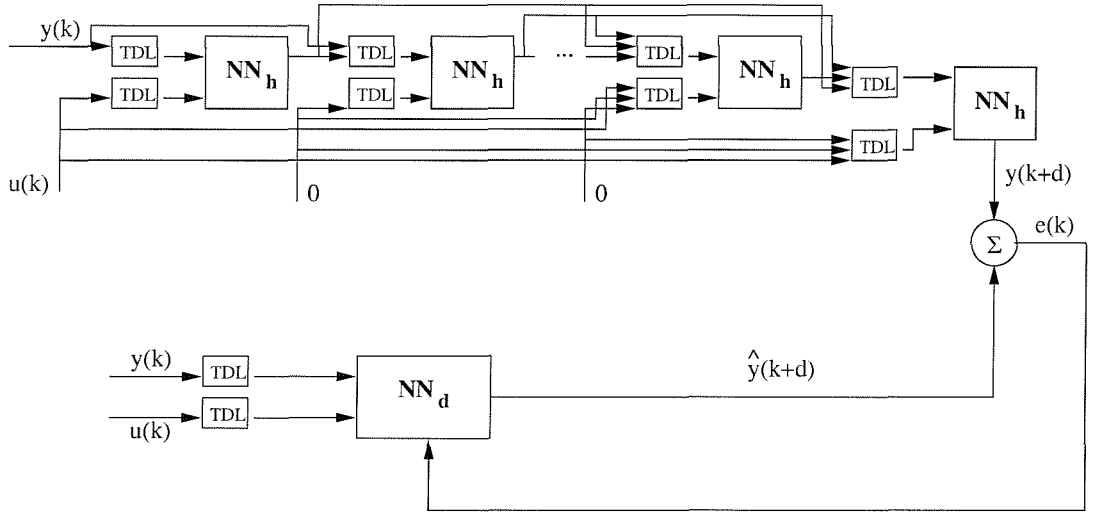


Figure 8.17: Training the network NN_d using the network NN_h

Results

The required networks were trained on an example in (Levin and Narendra 1996). The third order system considered is defined by:

$$
\begin{aligned}
x_1(k+1) &= 0.5x_3(k) \\
x_2(k+1) &= x_1(k) + [1 + 0.4x_2(k)]u(k) \\
x_3(k+1) &= 0.3x_1(k)x_3(k) - x_2(k) \\
y(k) &= x_1(k)
\end{aligned} \tag{8.16}
$$

Figure 8.18 shows the trajectory of the network $NN_h \in NN^3_{5,10,5,1}$ after convergence. The output of the plant has been normalised such that $y \in (-3, 3)$ and the training parameters $\eta = 0.55$ and $\alpha = 0.95$ were used.

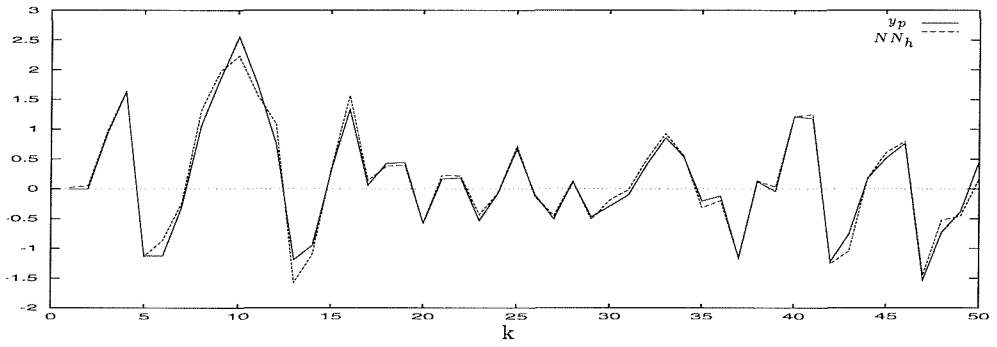


Figure 8.18: Identification of y(k+1)

Figure 8.19 shows the trajectory of the network $NN_d \in NN^3_{5,10,5,1}$ as an estimate of $y(k+d)$ compared to the estimate of $y(k+1)$ as provided by network NN_h. The network was trained using the same training parameters as with NN_h. Both networks converged after about 30000 steps.

8.2.3 Stabilisation and Tracking

In this section it is assumed that a sufficiently accurate model - supplied independently or obtained by system identification - is available. The particular form of the model used will depend on the control problem considered, i.e regulation or

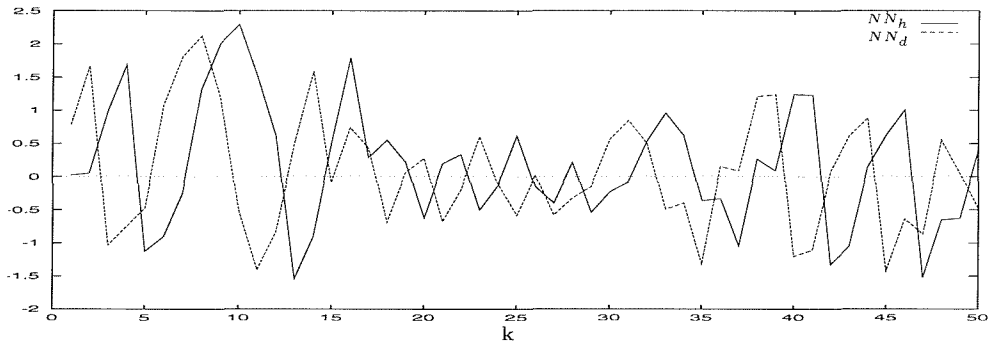


Figure 8.19: Prediction of y(k+d)

tracking.

In the case of regulation, the objective is to stabilise the system about a specified equilibrium point using only input and output measurements. By definition, a state space model of the plant is assumed to exist and the regulation problem consists of two steps.

1. Estimation of the states from input-output observations.
2. Design of a feedback control law using the estimated states.

The state estimation problem has been treated earlier in section 7.3.1. The additional theory required here for the second problem is given by the following result from (Levin and Narendra 1993).

Theorem 5 *Consider the nonlinear system state space model (7.1) with (7.18) describing its linearisation around the origin. Suppose also that (7.18) is controllable and observable. Then locally there exists an output feedback law*

$$u(k) = g[Y_n(k-n+1), U_{n-1}(k-n+1)] \tag{8.17}$$

that will stabilise this linear system around the origin.

Network implementation consists of two phases that can be undertaken independently and the training procedure is the same for both cases, i.e. assumed

availability of a model or its identification from input output data. In both cases, the actual training is done on a model and the resulting controller could then be applied to the plant. The training of the observer has been detailed earlier in this chapter and once it is complete, it can be assumed that a neural network NN_Φ (given the proper number of past input and output data) will generate as its output an accurate estimate of the current state of the system, i.e.

$$\hat{x}(k) = NN_\Phi[Y_n(k-n+1), U_n(k-n+1)] = x(k) + e_x(k) \tag{8.18}$$

In this last equation, $e_x(k)$ is the state estimation error where $||e_x(k)|| < \epsilon \ll 1$ for all k.

A state feedback based controller can be trained independently off-line on a model of the system it is to stabilise and, since controller training is done on a model of the system, it can be assumed that the states of the latter are accessible. The training of a controller NN_g to stabilise the system about the origin will now be defined, starting with the following result (Levin and Narendra 1993).

Theorem 6 *Consider the nonlinear system (7.1) and its linearisation about the origin (7.18). Then if the linearised system is controllable there exists a neighbourhood $V_x \subset X$ around the origin and a continuous feedback law $u(k) = g[x(k)]$ that will make V_x n-step stable with respect to the origin, i.e. any point $x_0 \in V_x$ can be driven to the origin in at most n steps.*

The existence of the deadbeat controller of the last theorem enables an objective function for the training procedure to be specified.

Now define $\overline{f}_g \equiv f[\cdot, g(\cdot)]$ and $\overline{F}_g \equiv \overline{f}_g^n(\cdot)$. Then by the last theorem, there exists an open set $V. \subset X$ containing the origin such that for all $x \in V$, $\overline{F}_g(x) = 0$. Also since $\overline{F}_g(\cdot)$ is a continuous function, it follows that there exists a larger open

set W containing V as a proper subset such that for all $x \in W$

$$||\overline{F}_g(x) - \overline{F}_g(0)|| \leq ||\overline{F}_g(x) - 0|| < ||x|| \tag{8.19}$$

Using the contraction mapping theorem, it now follows that for any $x \in W$, $\lim_{k\to\infty} \overline{F}_g^k(x) \equiv \overline{f}_g^{kn}(x) = 0$, and, for a given x, k is finite. It is this fact that is exploited in the design of a nonlinear controller for (7.1) to yield closed loop stability in a finite number of steps. The design objective is to choose $g(x)$ to make W as large as possible.

For a given $g(x)$, define the autonomous dynamical system

$$\overline{x}(k+1) = \overline{F}_g[\overline{x}(k)] \tag{8.20}$$

Then it follows that $V(x) = ||\overline{x}||$ is a Lyapunov function for this system for all $\overline{x} \in W$. To make W as large as possible, g must be adjusted such that $||\overline{x}(k+1)|| < ||\overline{x}(k)||$. As will be established below, this is achieved naturally using neural networks.

In the method described thus far, the region W depends on the system and is at least as big as that obtained using a linear controller. Also the existence of a local deadbeat controller that stabilises the system around the origin effectively establishes the following result.

Theorem 7 *Let Z denote the set controllable to the origin. Then there exists a feedback control law $g : X \to U$ that makes Z finitely stable with respect to the origin.*

The control law above is global but need not be continuous and hence it is not clear how it can be implemented using continuous neural networks. This problem

has been the subject of some work (Sontag 1990) but here attention is restricted to continuous control laws and next learning methods that optimize the range over which the contraction stabilising controller is valid are given.

For the above analysis to be applicable in a given case, the controllability of the linearisation must be checked. To do this, first determine the Jacobian of NN_f with respect to the inputs at the equilibrium points and thereby define

$$\hat{A} = \frac{\partial NN_f(x,u)}{\partial x}|_{0,0}, \quad \hat{B} = \frac{\partial NN_f(x,u)}{\partial u}|_{0,0} \tag{8.21}$$

and check if $\{\hat{A}.\hat{B}\}$ is a controllable pair by any of the standard tests. Also let S denote the region of which stabilisation is desired.

The task now is to train a neural network NN_g as a controller for (7.1) such that S is finitely stable with respect to the origin. The results given earlier establish the existence of a control law $u = g(x)$ for which the following are true.

1. There exists an open set V containing the origin such that for all $x \in V$, $\overline{F}(x) = 0$.

2. There exists a larger open set W (V is a proper subset) such that for all $x \in W$, $\overline{F}(x)$ is a contraction mapping.

Using these results, the performance of a controller can only be evaluated in n-step intervals and it is assumed that $u = g(x)$ can be found such that W covers S. The controller training is done using the model and therefore arbitrary initial conditions can be used which are selected using a random distribution over S. Now let

$$NN_{f,g}(x) = NN_f[x, NN_g(x)] \tag{8.22}$$

Then, once an initial point x_0 is chosen, $x_n = NN^n_{f,g}(x_0)$ is calculated by running

the controlled model through n steps.

It is only for the unknown $x \in V$ that the system can be driven to zero in n steps, the training error for the controller must be chosen as

$$e(x) = \begin{cases} x_n, & \text{if } ||x_0|| < \rho \text{ or} ||x_n|| > \lambda ||x_0|| \\ 0, & \text{otherwise.} \end{cases} \tag{8.23}$$

Here $\rho > 0$ is initially chosen to be 'small', and the parameter $0 < \lambda < 1$, which determines the contraction over W, initially chosen 'close' to 1. Next the actual implementation of this approach is detailed.

Network Implementation: Once stabilizability is confirmed the training is to be set up so as to provide stabilisation from input/output measurements as shown in figure 8.20. Training of the stabiliser is performed using the model of the system

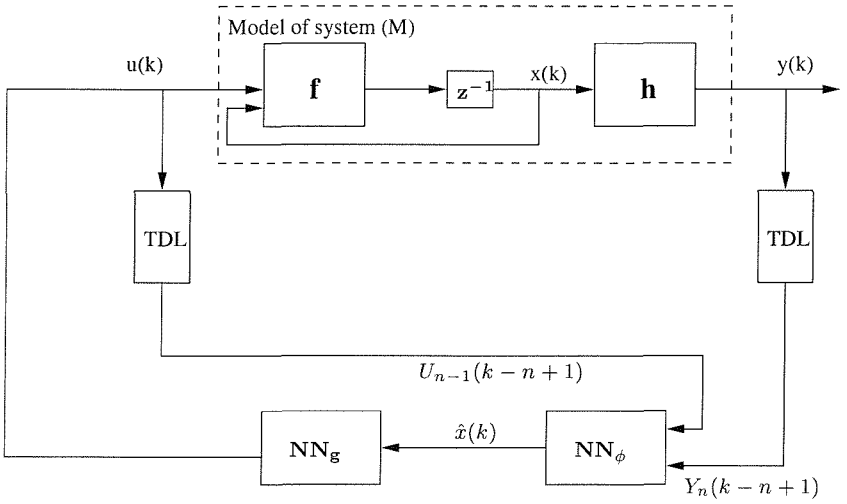


Figure 8.20: Stabilization using input / output measurements (Levin and Narendra 1996)

and therefore arbitrary initial conditions can be assumed (randomly chosen from S). Once the initial point x_0 is chosen $x_n = NN^n_{f,g}(x_0)$ is calculated, as defined in (8.22). This is represented diagramatically in figure 8.21 and the training error is calculated using (8.23).

The error gradient calculation is not as simple (relatively) as in the previous section. The problem is that the error cost function (when $e(x) \neq 0$):

$$E = \frac{1}{2}\sum_{m}(0 - \hat{x}_m)^2 \tag{8.24}$$

is not directly related to the output of the neural network NN_g. This requires an extra term in the chain rule:

$$\Delta w_{ij}^{(L-1)} = \eta \frac{\delta E}{\delta \hat{x}_m} \frac{\delta \hat{x}_m}{\delta u_j} \frac{\delta u_j}{\delta w_{ij}^{(L-1)}} \tag{8.25}$$

where $\mathbf{u} = NN_g(\hat{\mathbf{x}}(k))$. In reality, as only SISO plants are considered, there is only one output node corresponding to j in the equation. However, the scheme is relatively easily extended to MIMO systems so the notation is retained.

In order for conventional error back propagation to be applied here, the system needs to be unfolded to take into account delays in the plant. The original scheme proposed by Levin and Narendra (1993) is shown in figure 8.21.

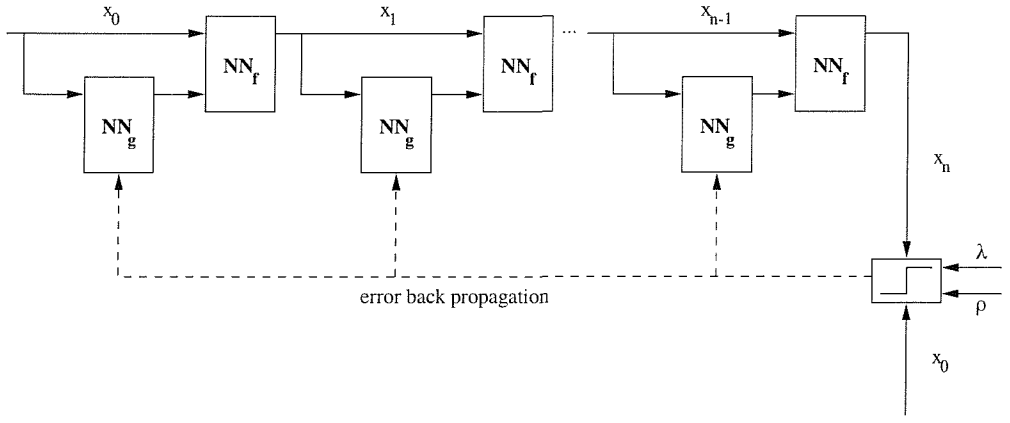


Figure 8.21: Training a direct stabiliser using state inputs (Levin and Narendra 1993)

where n denotes the order of the plant.

This particular approach uses the trained network NN_f as an observer, with

the added advantage of requiring no additional plant model for training, since the state estimates of NN_f are used as its next input. However, the disadvantage with such a method is that any error in estimation will become compounded. It is also desirable to provide stabilisation from just past input and output data. An alternative unfolding of the system for such a method is shown in figure 8.22.

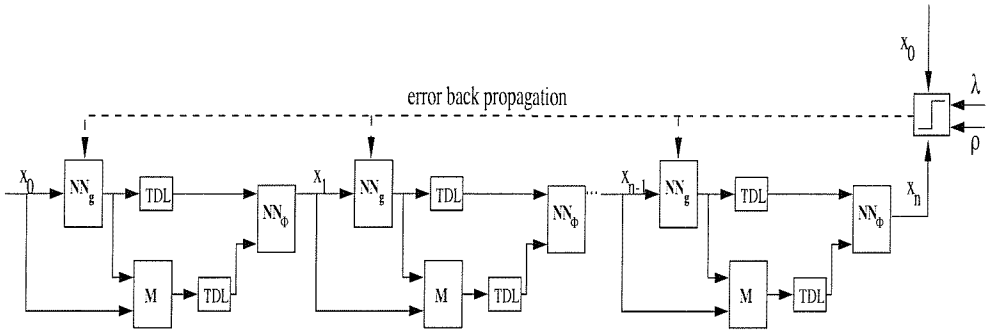


Figure 8.22: Training a direct stabiliser using input-output data

If figure 8.22 is used, its execution can be greatly simplified by training the stabiliser using the plant model states (figure 8.23). This does not require any further assumptions about the system and is liable to produce more accurate results. The stabilisation of the scheme during run time will still be achieved using the state estimates from observer network NN_Φ as in figure 8.20.

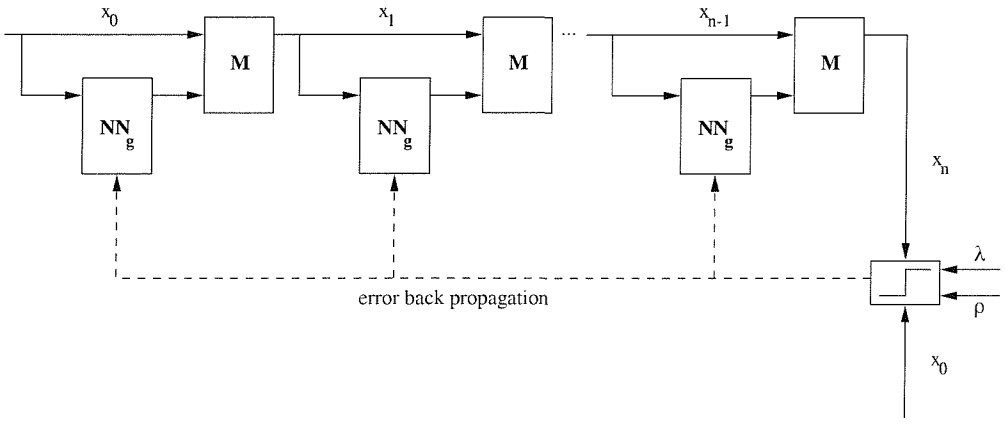


Figure 8.23: Training a stabiliser direct from the model state inputs

As can be seen from the figures, a weight change needs to occur for each

controller stage in the unfolding. The error back propagation becomes:

$$\Delta w_{ij}^{(L-1)} = \sum_{m}\sum_{t=0}^{n-1} \eta \frac{\delta E}{\delta \hat{x}_m(t)} \frac{\delta \hat{x}_m(t)}{\delta u_j(t)} \frac{\delta u_j(t)}{\delta w_{ij}^{(L-1)}} \tag{8.26}$$

or with $u_j(t) = y_j^{(L)}$:

$$\Delta w_{ij}^{(L-1)} = -\sum_{m}\sum_{t=0}^{n-1} \eta \hat{x}_m \frac{\delta \hat{x}_m(t)}{\delta u_j(t)} (1 - u_j(t)) u_j(t) y_i^{(L-1)} \tag{8.27}$$

and $\frac{\delta \hat{x}_m(t)}{\delta u_j(t)}$ is estimated using the Taylor series approximation:

$$\Psi'(x) \approx \frac{\Psi(u_j + h)) - \Psi(u_j)}{h} \tag{8.28}$$

where Ψ is the relevant section of the unfolding in figure 8.21 or 8.22.

Necessary Alterations

Scaling Factors on the Input Values

In order to aid convergence of a solution with state inputs that lie in small ranges, the effective difference between inputs is increased by multiplying each input by a scaling factor.

Repeating Points

The objective of the stabiliser was to stabilize the system inside S, $S = \{x| \, ||x|| \leq 2\}$. W was initially chosen as $W = \{x| \, ||x|| \leq 0.1\}$. It was found through trial and error that convergence was not possible where random points in W were encountered only once. In order to improve convergence of the stabiliser network, each point was presented up to t_l times. A new point was chosen when the current point had been contracted or after t_l presentations. Training was performed using the system in figure 8.23. The network $NN_g \in NN^3_{3,10,5,1}$ took

284642 time steps to converge with $\alpha = 0.95$, $\eta = 0.55$, $t_l = 35$, $\lambda = 0.95$, $\rho = 0.01$ and with the outputs of the network normalised within $u \in (-5, 5)$ and the state inputs were normalised with a multiplying factor of 1000. Figure 8.24 shows the stabilisation of a point $\mathbf{x} = (0.01, 0.01, 0.01)$ by NN_g with estimated state inputs from NN_Φ as represented in figure 8.8.

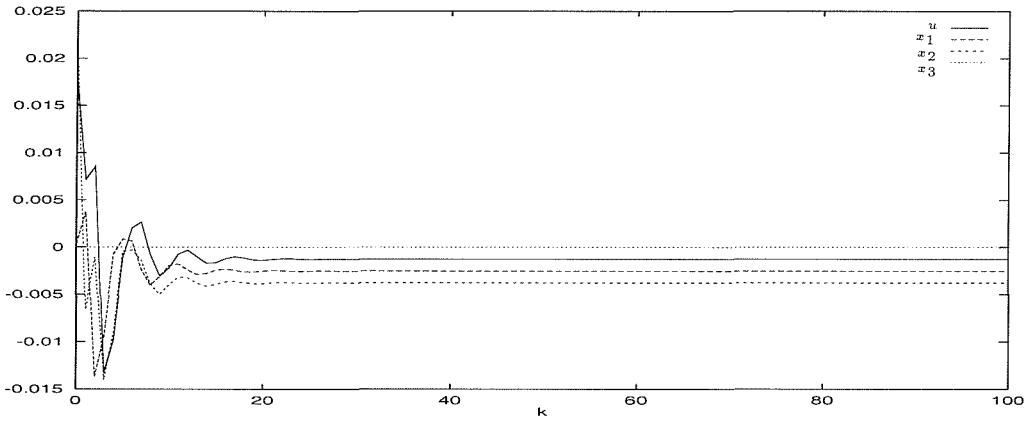


Figure 8.24: Stabilization of point $\mathbf{x} = (0.01, 0.01, 0.01)$ with NN_g trained using repeated points

Figure 8.25 shows the limits of the trained stabiliser with the stabilisation of point $\mathbf{x} = (0.0380, 0.0413, 0.0299)$. This is equivalent to $||\mathbf{x}|| = 0.0636$ and no point outside this range is stabilizable by the network NN_g.

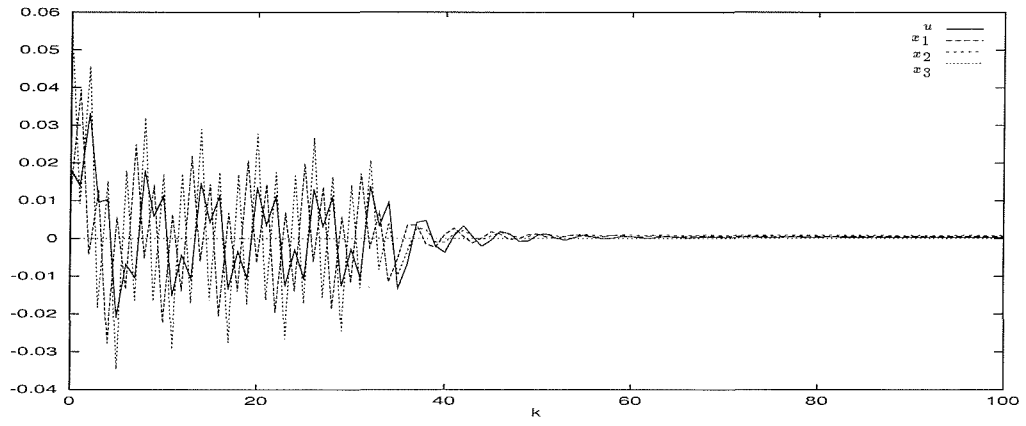


Figure 8.25: Stabilization of point $\mathbf{x} = (0.0380, 0.0413, 0.0299)$ with NN_g trained using repeated points

Driving Points to zero

Although the utilisation of repeated points helped with convergence for examples involving a small area round the origin, examples with ranges outside $||x|| = 0.1$ were extremely slow to converge. An added component of training was introduced with the aim of driving all points to zero. This is not possible for all points, but whenever a point contracted successfully, the new point was chosen as the contracted point rather than as a new random point in W. An early result of this is shown in figure 8.26. Here the stabiliser is attempting to stabilize the point $\mathbf{x} = (0.085, 0.085, 0.085)$. All training parameters are the same as in the previous example.

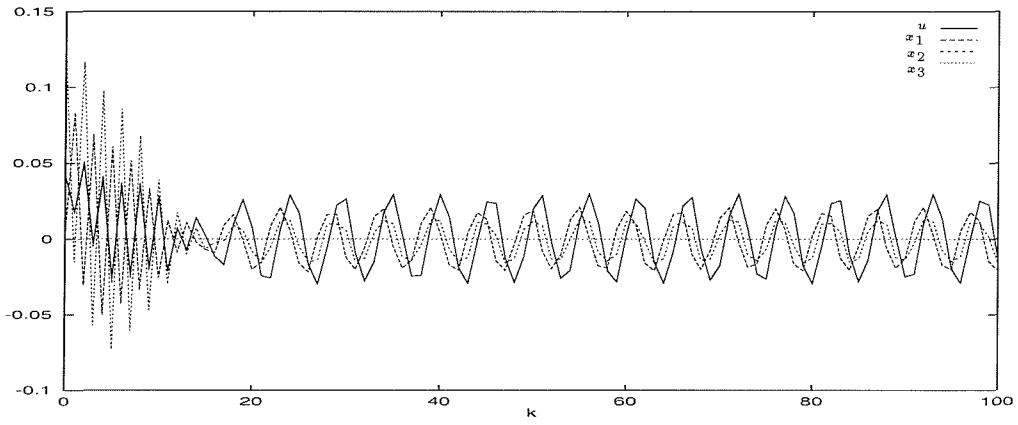


Figure 8.26: Attempted stabilisation of point $\mathbf{x} = (0.085, 0.085, 0.085)$ with NN_g trained using repeated points and driving points to zero

Although the stabilisation is not successful in figure 8.26, the result is a promising one. Having taken 202991 steps to converge, the convergence rate is faster and the point is well outside the range of the previous examples ($||\mathbf{x}|| = 0.147$). The oscillation could be due to the bias that exists in this training method towards points outside of V (estimated as ρ). This can be overcome by reducing the value of ρ causing each point to be contracted nearer to zero and reducing the bias.

Variable Training Parameters

To further aid convergence, a variable training parameter was introduced:

$$\Delta w_{ij}^{(L-1)} = k\eta||\mathbf{x}_0||\frac{\delta E}{\delta \hat{x}_m}\frac{\delta \hat{x}_m}{\delta u_j}\frac{\delta u_j}{\delta w_{ij}^{(L-1)}} \tag{8.29}$$

Figure 8.27 shows the control action NN_g when trained to stabiliser points in W, $W = \{x|\,||x|| \leq 0.5\}$. The training parameters are the same as before with the exception that here $\rho = 0.005$, $k\eta = 3.5$ and the multiplying scaling factor on the input has been reduced to 100. The network converged after only 37169 steps and the figure shows the stabilisation of a point $\mathbf{x} = (0.25, 0.25, 0.25)$, $||\mathbf{x}|| = 0.43$.

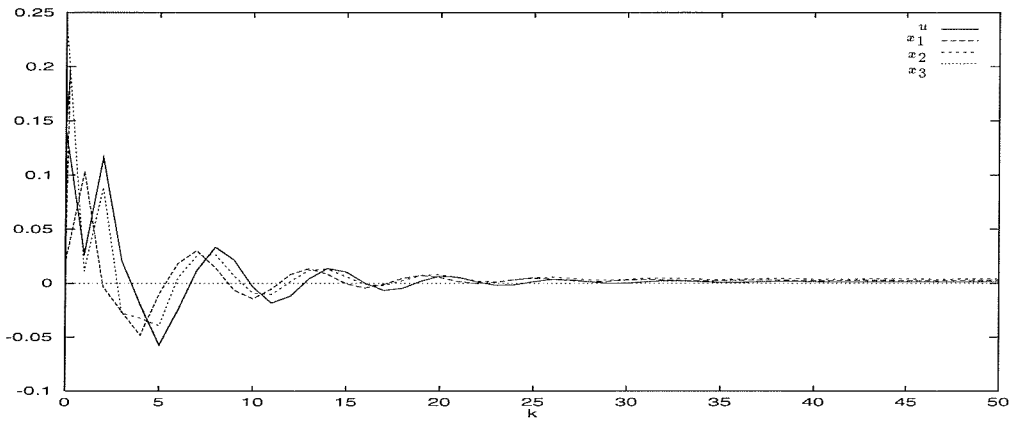


Figure 8.27: Stabilization of point $\mathbf{x} = (0.25, 0.25, 0.25)$ with NN_g trained using a variable training parameter

Figure 8.28 shows the control action of NN_g when trained to stabilize points in W, $W = \{x|\,||x|| \leq 1.0\}$. The training parameters are the same as previous with the exception that here $k\eta = 0.5$ and a scaling factor of 20 is used. Oscillation is present here, which in the example in figure 8.26 was cured by a reduced value of ρ. Figure 8.29 shows an example where $\rho = 0.001$. The oscillation still present. This example highlights a problem which occurs for larger choices of W in that the scaling factor used can hinder convergence if too large a value is selected.

Figure 8.30 shows an example where the scaling factor has been reduced to 19.5, $\rho = 0.005$ and $k\eta = 0.49$ in this example.

All that remains is to train a tracking controller that can control the plant to follow a model of reference. In other words, a network NN_c is to be trained to emulate the mapping:

$$u(k) = c[Y_n(k-n+1), U_{n-1}(k-n+1), \overset{*}{y}_m(k+d)] \tag{8.30}$$

that will cause the system to follow the reference model $\overset{*}{y}_m(k)$

Network Implementation: The system is set up as shown in figure 8.31.

A controller can be trained for a plant where the states are unavailable in two ways. Either the state input in figure 8.31 can be estimated using the observer network NN_Φ and all else remains unchanged; or the controller can be trained off input output data directly, as shown in figure 8.32.

In similar fashion to the stabiliser network, the weights are adjusted as a function of the response of the network NN_d to the controller input. Equivalently, the error cost function

$$E = \frac{1}{2}\sum_m (\overset{*}{y}_m(k+d) - \hat{y}_m(k+d))^2 \tag{8.31}$$

is used to adjust the weights according to (with $u_j(k) = y_j^{(L)}$):

$$\begin{aligned} \Delta w_{ij}^{(L-1)} &= \sum_m \frac{\delta E}{\delta \hat{y}_m(k+d)} \frac{\delta \hat{y}_m(k+d)}{\delta u_j(k)} \frac{\delta u_j(k)}{\delta w_{ij}^{(L-1)}} \\ &= \sum_m (\overset{*}{y}_m(k+d) - \hat{y}_m(k+d)) \frac{\delta \hat{y}_m(k+d)}{\delta u_j(k)} \times \\ &\qquad u_j(k)(1-u_j(k)) y_i^{(L-1)} \end{aligned} \tag{8.32}$$

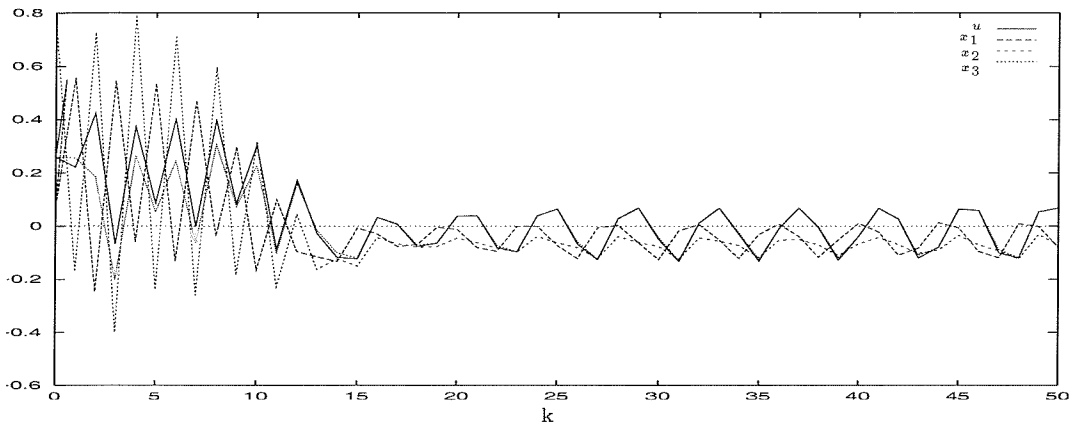


Figure 8.28: Attempted stabilisation of point $\mathbf{x} = (0.55, 0.55, 0.55)$ with $\rho = 0.005$ and a scaling factor = 20

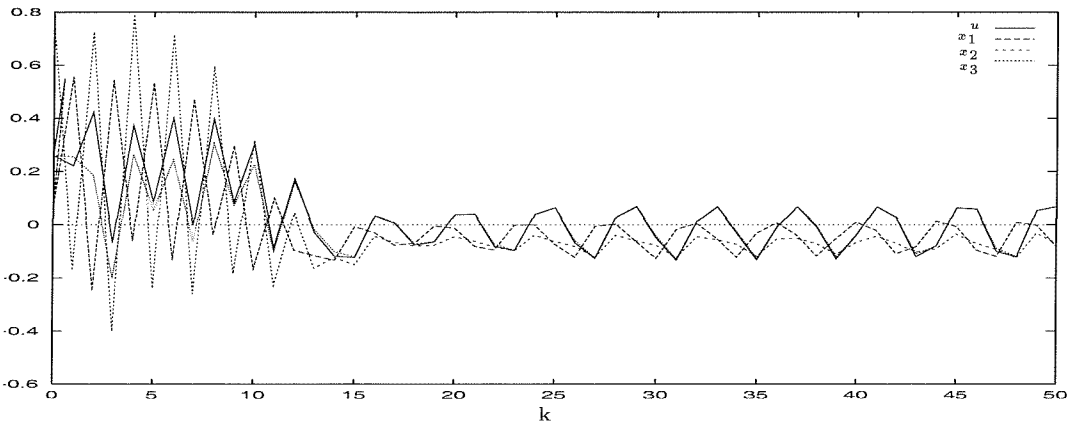


Figure 8.29: Attempted stabilisation of point $\mathbf{x} = (0.55, 0.55, 0.55)$ with $\rho = 0.001$ and a scaling factor = 20

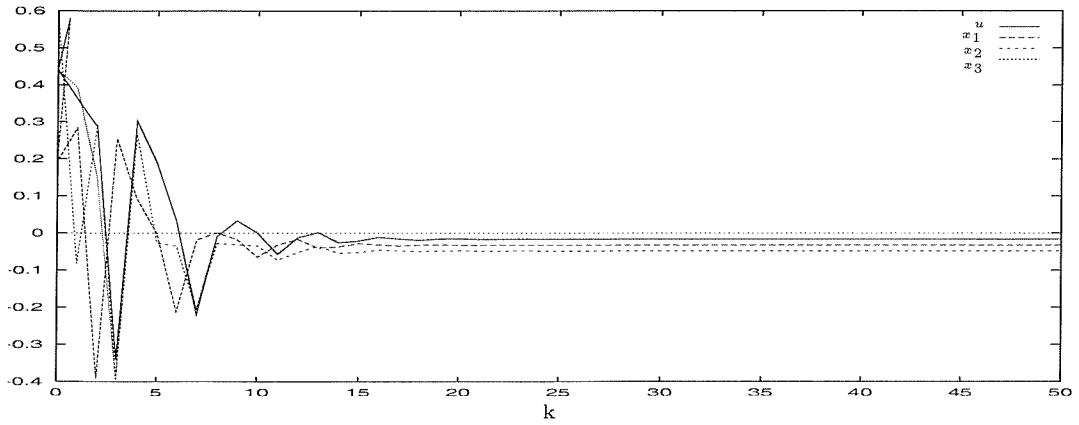


Figure 8.30: Stabilization of point $\mathbf{x} = (0.58, 0.58, 0.58)$ with $\rho = 0.005$ and a scaling factor = 19.5

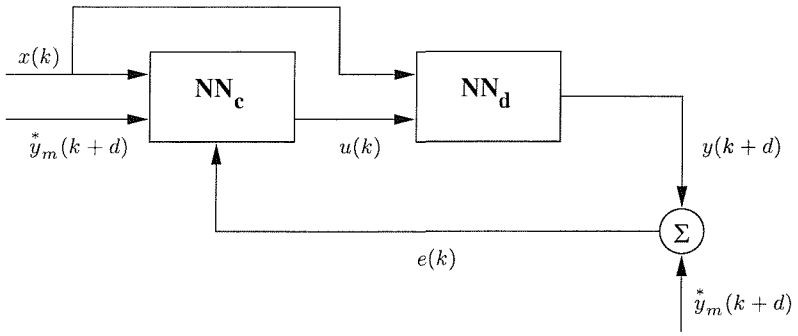


Figure 8.31: Training the network NN_c

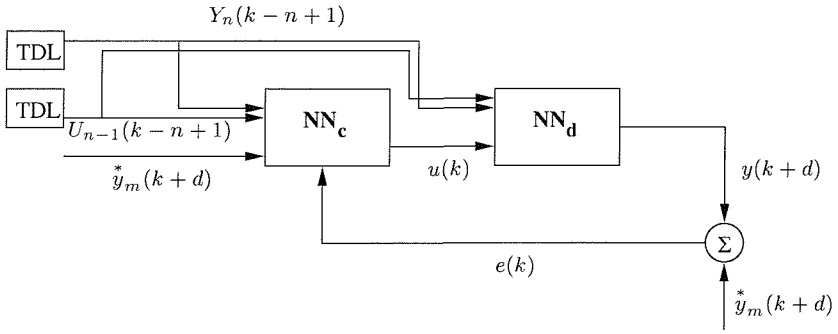


Figure 8.32: Training the network NN_c off input output data directly

where $\frac{\delta\hat{y}(k+d)}{\delta u(k)}$ is estimated using the Taylor series approximation:

$$\Psi'(x) \approx \frac{\Psi(u_j + h)) - \Psi(u_j)}{h} \tag{8.33}$$

and Ψ represents $y_m(k + d)$.

Results

The network $NN_c \in NN^3_{6,10,5,1}$ was trained to control the output of the system described in (8.16) to follow the reference signal $y_m = sin(2\pi(k + 2)/7.5)$. The network was trained using a fixed training parameter $\eta = 0.02$ and a momentum $\alpha = 0.85$. Convergence was achieved after 75000 steps. The result is shown in figure 8.33.

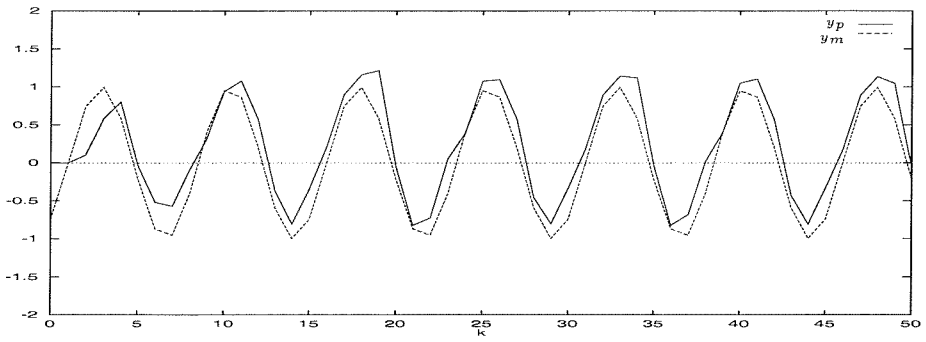


Figure 8.33: Controlling the system y_p to follow a reference trajectory where $y_m = \overset{*}{y}_m(k+d)$

8.3 A Multiple Model Based Adaptive Control Scheme Based on Neural Network Models

In this section, the implementation of the multiple model based adaptive control scheme of chapter 6 is considered based on neural network models. Figure 8.34 shows a schematic of this scheme where, as in chapter 6, a number of fixed models operate alongside a real-time adaptive identification model. Model quality is evaluated using the cost function:

$$J_i(t) = \alpha\epsilon_i^2(t) + \beta\int_0^t e^{-\lambda(t-\tau)}\epsilon_i^2(\tau)d\tau \tag{8.34}$$

Again a hysteresis switching component, δ, is built into the switch such that a new model is chosen if the following condition is met:

$$J_{new} + \delta < J_{cur} \tag{8.35}$$

The adaptive component here is quite different from that of chapter 6. With the training of the controller requiring a ready trained predictor $y(k+d) = NN_d$,

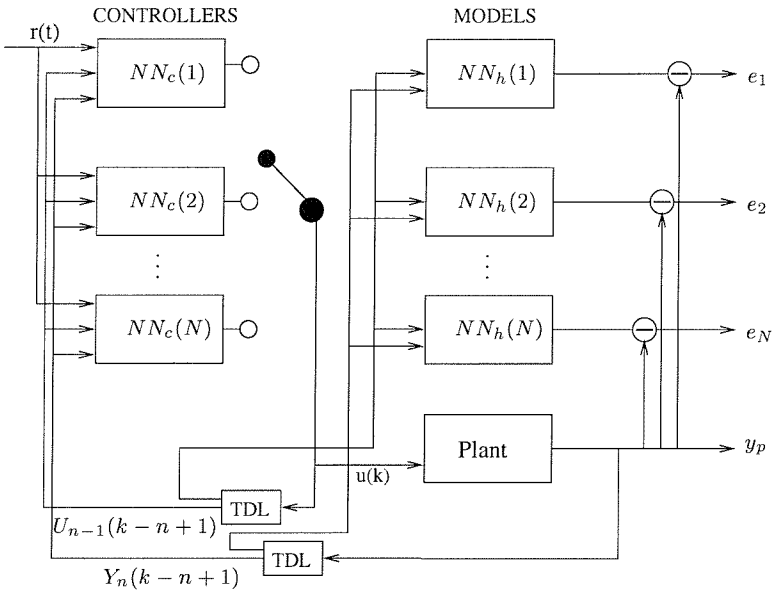


Figure 8.34: A revised switching scheme incorporating neural networks

it is no longer possible to use a completely free adaptive model. This problem can be overcome by resetting the adaptive network weights to the best model each time a new model is chosen. This requires that the networks $NN_h(new)$, $NN_d(new)$ and $NN_c(new)$ be communicated whenever a new model is chosen. This is highly inefficient, so to reduce the overheads, all fixed networks are communicated to the adaptive processor (1) before the scheme is started. This means that any fixed model can be identified by a single integer tag during run time.

8.4 Parallelisation of the Scheme

The Training Phase

The scheme requires the availability of trained networks NN_h, NN_d and NN_c for each plant environment. Narendra *et al.* (1995) advocate the automatic detection of differin environments during the training process. However, as an explicit model of the plant is required during each stage of the training process, the use-

fulness of this approach is limited. Assuming detailed knowledge of each plant environment allows the training process to be parallelised by distributing information of the plant environments evenly amongst the available processors.

In order to reduce communication overheads, the run time topology is used for allocation of networks for the training phase. Therefore, processor 0 is reserved for the master and processor 1 for the adaptive component where possible. This means that the master and adaptive processors remain idle during the training process. The trained networks are then communicated to the master and stored for future runs of the switching scheme.

The Switching Scheme

The switching scheme shown in figure 8.34 is parallelised in exactly the same way as the linear case shown in figure 6.6. The basic structure is identical with the sequential tasks of simulation and sampling of the plant sandwiching the parallel task of calculating the model outputs (see figure 6.6b).

The processor topology is as shown in figure 8.35. The master processor is reserved for plant simulation, sampling and performs the final switch in the switching scheme. The adaptive processor performs fine tuning of the fixed models run on the other worker processors.

The Communication Strategy

The communication strategy has changed somewhat relative to, chapter 6 in an effort to minimise the communication of the large amount of data required to represent each neural network (compared to a few numbers representing the linear parameter space in the previous chapter). The tree topology is as before and is shown in figure 8.35.

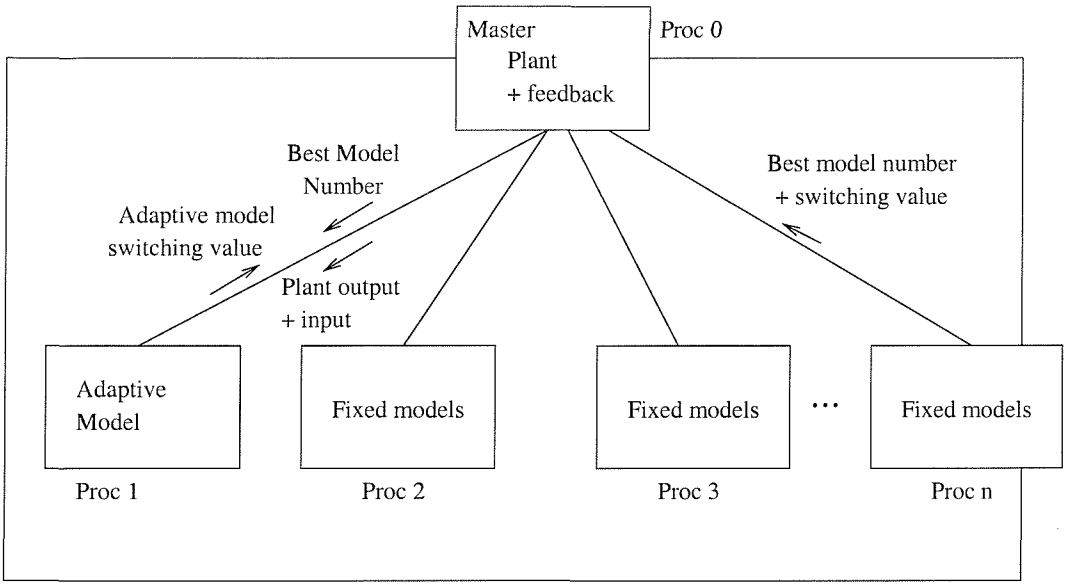


Figure 8.35: Tree topology with the communication strategy for the revised switching scheme.

Set-up Communications

To reduce run time communications, the master sends the data of all the fixed NN_h, NN_d and NN_c networks to the adaptive processor. As training is only required once per plant example, it is desirable to allow the master to load network data from disk, and in these cases the master has to distribute model data amongst the worker processors available. The pattern of distribution is shown in the tables in section 8.6.

The run time data is divided into three packets:

Sample data - *From the master to the workers.*

Details of input and output plant data for model calculation purposes, together with the reference signal value for controller purposes. The final component of this packet contains a best processor flag so that each processor can determine if the best model has come from them and introduce hysteresis to local switching if required.

Model data - *From the workers to the master.*

Details of best model number, the cost function value of the best model and the control input corresponding to this best model. Note that since the plant is nonlinear, no averaging is performed because the theories that allow interpolation between models in the linear scheme are invalid in the nonlinear case.

Best Model Data - *From the master to the adaptive processor.*

This consists of an integer tag identifying the best model. Since the adaptive processor stores all the fixed models, no further communication is required by the master.

8.5 Results and Discussion

A six environment plant was set up as in equations (8.36) to (8.41).

Environment 1:

$$\begin{aligned} x_1(k+1) &= 0.5x_3(k) \\ x_2(k+1) &= x_1(k) + [1 - 0.4x_2(k)]u(k) \\ x_3(k+1) &= 0.3x_1(k)x_3(k) - x_2(k) \\ y(k) &= x_1(k) \end{aligned} \tag{8.36}$$

Environment 2:

$$\begin{aligned} x_1(k+1) &= 0.3x_3(k) \\ x_2(k+1) &= x_1(k) + [1 - 0.25x_2(k)]u(k) \\ x_3(k+1) &= 0.5x_1(k)x_3(k) - x_2(k) \\ y(k) &= x_1(k) \end{aligned} \tag{8.37}$$

Environment 3:

$$\begin{aligned} x_1(k+1) &= 0.3x_3(k) + 0.3x_1(k) \\ x_2(k+1) &= x_1(k) + [1 - 0.5x_2(k)]u(k) \\ x_3(k+1) &= 0.4x_1(k)x_3(k) - x_2(k) \\ y(k) &= x_1(k) \end{aligned} \tag{8.38}$$

Environment 4:

$$\begin{aligned} x_1(k+1) &= 0.3x_3(k) + 0.2x_1(k) \\ x_2(k+1) &= x_1(k) + [1 - 0.4x_2(k)]u(k) \\ x_3(k+1) &= 0.3x_1(k)x_3(k) - x_2(k) \\ y(k) &= x_1(k) \end{aligned} \tag{8.39}$$

Environment 5:

$$\begin{aligned} x_1(k+1) &= -0.3x_3(k) \\ x_2(k+1) &= x_1(k) + [1 - 0.2x_2(k)]u(k) \\ x_3(k+1) &= 0.3x_1(k)x_3(k) - x_2(k) \\ y(k) &= x_1(k) \end{aligned} \tag{8.40}$$

Environment 6:

$$\begin{aligned} x_1(k+1) &= 0.3x_3(k) \\ x_2(k+1) &= x_1(k) + [1 - 0.35x_2(k)]u(k) \\ x_3(k+1) &= 0.4x_1(k)x_3(k) - x_2(k) \\ y(k) &= x_1(k) \end{aligned} \tag{8.41}$$

Environment changes occurred after every 100 time steps. The following cost function was chosen with an hysteresis constant $\delta = 0.5$.

$$J_i(t) = 100\epsilon_i^2(t) + 200\int_0^t e^{-0.5(t-\tau)}\epsilon_i^2(\tau)d\tau \tag{8.42}$$

The networks were chosen, as in previous sections, with $NN_h \in NN^3_{5,10,5,1}$, $NN_d \in NN^3_{5,10,5,1}$ and $NN_c \in NN^3_{6,10,5,1}$. The training parameters were chosen as: $\eta = 0.55$ and $\alpha = 0.95$ for the NN_h and NN_d networks, and $\eta = 0.018$ and $\alpha = 0.85$ for the NN_c networks. The reference model was chosen as $y_m = sin(2\pi(k+2)/7.5)$.

The results are shown in figure 8.36. The error in the control of the first plant environment ($k \in [0 : 100)$) is unsatisfactory. This is due to the wrong model being chosen by the scheme and is in turn due to the transient error caused by the delay in the plant. The model that gives the lowest output (closest to the actual plant output of zero for the first $d = 3$ time steps) is chosen. The third plant environment ($k \in [200 : 300)$) is also fairly poor, although this is due in part to a poor convergence of the neural networks that provide identification and control for this environment (relative to the other models). This highlights a problem with using the same training parameters for the training of each model. However, if a unique training parameter is chosen for each model, it creates a problem with the choice of parameters for the training of the adaptive component.

In an attempt to overcome the problems of control of the initial environment, a cost function with greater sensitivity to changes in error was chosen:

$$J_i(t) = \epsilon_i^2(t) + 6 \int_0^t e^{-0.65(t-\tau)} \epsilon_i^2(\tau) d\tau \tag{8.43}$$

The results for this cost function are shown in figure 8.37. The control of the first environment has been greatly improved. However, this has been achieved at the expense of accuracy in the other environments. The control is likely to still be acceptable, and the choice of cost function, along with the choice of training parameters and the hysteresis constant δ is unique to each problem and control requirement.

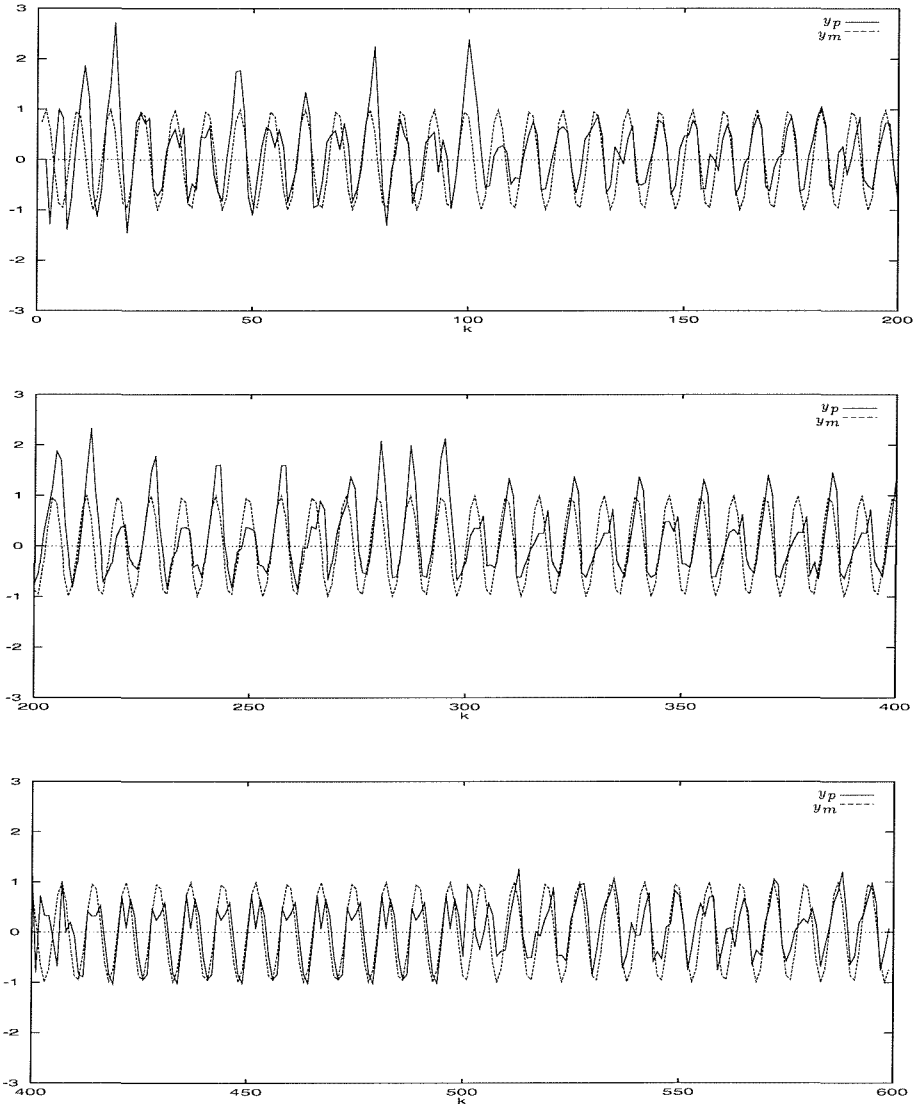


Figure 8.36: Control of a nonlinear plant with dynamic changes every 100 time steps using cost function $J_i(t) = 100\epsilon_i^2(t) + 200\int_0^t e^{-0.5(t-\tau)}\epsilon_i^2(\tau)d\tau$

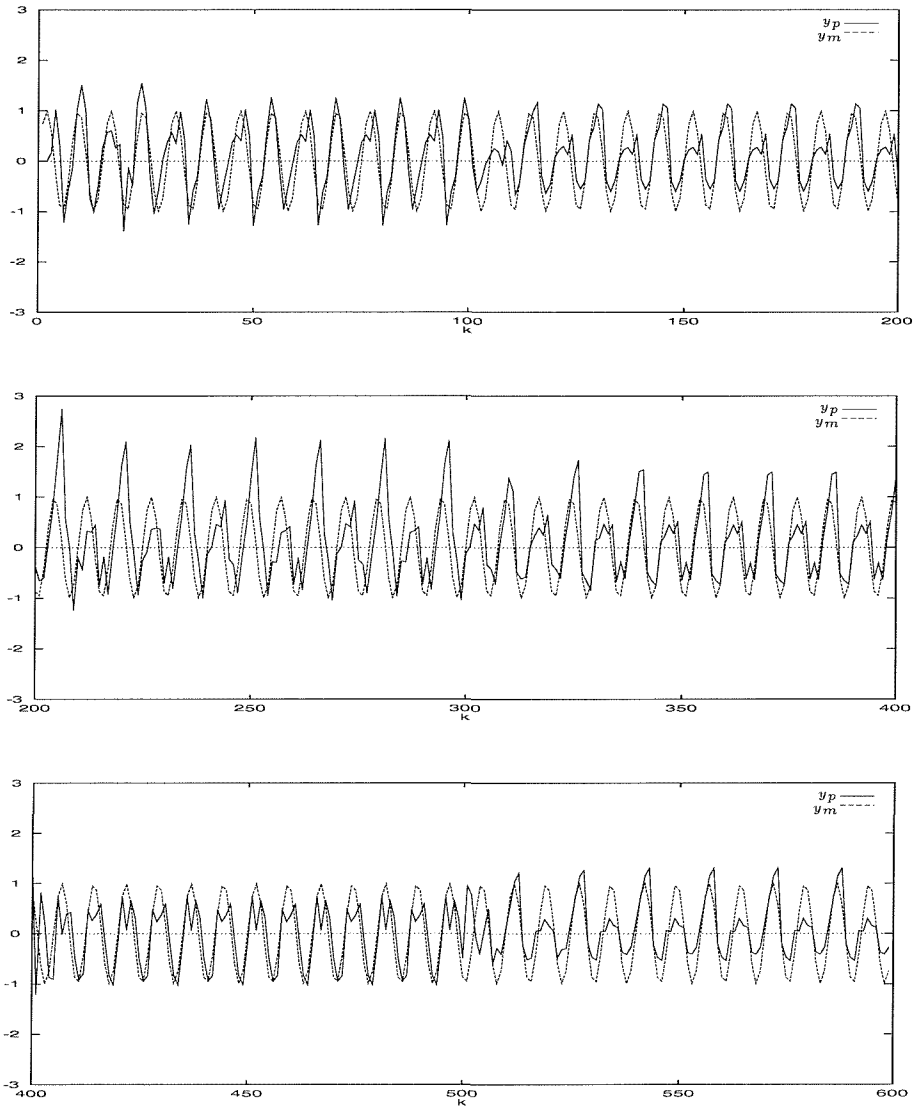


Figure 8.37: Control of a nonlinear plant with dynamic changes every 100 time steps using cost function $J_i(t) = \epsilon_i^2(t) + 6\int_0^t e^{-0.65(t-\tau)}\epsilon_i^2(\tau)d\tau$

8.6 Performance Analysis

8.6.1 Results on the PowerPC network

Table 8.1 shows the timings for the training process. As can be seen by the reduction in time between the sequential process (1×1 partition) and the 8 processor solution (2×4 partition), the training process is highly parallel. This is underlined by the scalability column which is an efficiency calculation defined by

$$Scalability = \frac{T_1}{NT_N} \tag{8.44}$$

where T_1 is the time taken to perform the calculation on a single processor and T_N is the time taken to perform the calculation over N processors.

The Scalability column settles down to a scalability of around 93%. The number of processors in the scalability calculation ignores processors reserved for the master and adaptive model as these processors remain idle during the training process. The time also ignores any time required to communicate the networks back to the master - these times will be the same as the fixed model communication times in table 8.2. These times include the time taken for the master to send all the fixed networks to the adaptive model processor and for the master to distribute the fixed models across the partition. These are one off set up times and so are insignificant since they will have no effect on the minimum length of the sampling period. This means that the times in table 8.1 for partitions 1×1, 1×2, 2×1 and 1×3 should all be the same and any anomally is due to memory access bottlenecks or nonoptimal caching which will be unique to each processor. The apparent drop in efficiency on the 2×3 partition is due to the non-uniform distribution of 6 models across 4 processors (the distribution is explicitly listed in the models per processor

column). This problem can be overcome by distributing the models evenly across six processors - ignoring the reservation for master and adaptive processors. This would require a redistribution of models after training. This is unlikely to affect the efficiency of the calculation since the communication time of distributing the networks across the processor topology is relatively small. This can be seen in table 8.2. For long runs this initial communication will become irrelevant to the overall run time, since it is a one-off-set-up overhead.

Table 8.3 shows the run times for the results shown in figures 8.36 and 8.37. The minimum sampling period for each partition is shown in the last column. This allows a comparison with speed up and scalability to be drawn. In order for a parallel solution to run faster than the sequential solution it is necessary for the following inequality to be met:

$$Scalability > \frac{1}{N} \tag{8.45}$$

This criteria is not met until the implementation on a 4 processor partition, where a speed-up of only 1.178 is achieved on the 1×4 partition and 1.212 on the optimal 2×2 partition (which is directly mapable onto the PowerXplorer PC component shown in figure 3.5. The speed-up on the 2×4 partition is only 1.416 but this is still equal to a minimum length of the sampling period that is about $\frac{7}{10}$'s the length of the sequential solution.

It is possible with the data available to produce a crude mathematical model of the processes involved. Defining a communication setup time as C, the fixed model calculation as F and the adaptive tuning process as A, the following equations -

Partition	No. of Fixed Networks on Processor ID Number								Time (s)	Scalability(%)
	0	1	2	3	4	5	6	7		
1 x 1	6	-	-	-	-	-	-	-	969.420	100.00
1 x 2	M	6	-	-	-	-	-	-	1046.838	92.60
2 x 1	M	6	-	-	-	-	-	-	1046.846	92.60
1 x 3	M	A	6	-	-	-	-	-	969.487	99.99
1 x 4	M	A	3	3	-	-	-	-	518.195	93.54
2 x 2	M	A	3	3	-	-	-	-	518.144	93.55
2 x 3	M	A	1	1	2	2	-	-	349.218	69.40
2 x 4	M	A	1	1	1	1	1	1	174.340	92.68

Table 8.1: Training times for 6 networks over various partitions (where possible, processor 0 is reserved for the master (M) and processor 1 for a real time adaptive model (A))

Partition	No. of Fixed Networks on Processor Number								Comms To Fixed Time (s)	Comms To Adapt Time (s)	Total Time (s)
	0	1	2	3	4	5	6	7			
1 x 1	6+A	-	-	-	-	-	-	-	0.0000	0.0000	0.0000
1 x 2	M	6+A	-	-	-	-	-	-	0.0518	0.0000	0.0518
2 x 1	M	6+A	-	-	-	-	-	-	0.0512	0.0000	0.0512
1 x 3	M	A	6	-	-	-	-	-	0.0593	0.0511	0.110
1 x 4	M	A	3	3	-	-	-	-	0.0647	0.0510	0.116
2 x 2	M	A	3	3	-	-	-	-	0.0550	0.0516	0.107
2 x 3	M	A	1	1	2	2	-	-	0.0612	0.0517	0.113
2 x 4	M	A	1	1	1	1	1	1	0.0648	0.0519	0.117

Table 8.2: Time taken to communicate trained networks to required processors

constructed from partitions 1×2, 2×2 and 2×4 - are arrived at

$$\begin{aligned} 1 \times 2 &: \quad A + 7W + C = 435 \\ 2 \times 2 &: \quad A + 3W + 3C = 282 \\ 2 \times 4 &: \quad A + W + 7C = 256 \end{aligned} \tag{8.46}$$

Note that the adaptive process is a sum of the fixed model calculation and the adaptive tuning process $(A + W)$ and that the model assumes a high message latency such that all messages take the same time to set-up for communication (C). The best model communication between the master and the adaptive processor is built into the quantity A. Solving the simultaneous equations of (8.46), gives the following solutions

$$\begin{aligned} A &= 91.2 \\ W &= 46.7 \\ C &= 16.9 \end{aligned} \tag{8.47}$$

which yield the following predictions

$$\begin{aligned} 1 \times 3 &: \quad A + 6W + 2C = 405.2 \\ 1 \times 4 &: \quad A + 3W + 3C = 282.0 \\ 2 \times 3 &: \quad A + 2W + 5C = 269.1 \end{aligned} \tag{8.48}$$

The predictions for partitions 1×3 (actually 411 secs) and 2×3 (actually 276 secs) are accurate, however the 1×4 partition is not. With a workload identical to the 2×2 partition, the 1×4 partition should yield the same result. The times are most likely to differ due to anomalies in the network routing strategy, but there is

no direct information in the documentation (Parsytec 1994) about this strategy. The 2×2 partition is an optimal topology for 4 processors, whereas the 1×4 partition can only be connected in a line across two connecting 2×2 processor components. This means, in communication terms, that the worst case message routing strategy passes through 1 processor (a total of 2 communications) in the optimal 2×2 partition and through 2 processors (a total of 3 communications) in the unoptimal 1×4 partition. There is also no attempt to model the sequential (1×1) process since the respective sequential and parallel codes differ to such an extent that they are not directly comparable.

Tables 8.4 - 8.6 show results for longer runs to test for consistency. The results do show some variation. However, since the variation is also observed in the sequential result, delays or inconsistencies exist in the computation area as well as the communication.

8.6.2 Results on the Transputer Network

As a direct comparison with the results on the PowerXplorer array, the switching scheme was run on the transputer array. Table 8.7 shows the set-up communication times to be used as a comparison with table 8.2. As can be seen, there is a consistency of times in the transputer communications that is not apparent in the PowerPC array. The communication times are also faster on the transputer array, despite the fact that both arrays use transputer links for communication. The transputer array has a backplane which allows 'direct' communication between processors on different boards whereas messages have to be routed through a number of processors on the PowerXplorer array.

Table 8.8 shows the run times for the switching scheme run over 600 time

Partition	No. of Fixed Networks on Processor Number 0	1	2	3	4	5	6	7	Time (s)	Scalability(%)	Min Length of Sampling Period(s)
1 x 1	6+A	-	-	-	-	-	-	-	2.265	100.00	0.00378
1 x 2	M	6+A	-	-	-	-	-	-	2.819	40.17	0.00470
2 x 1	M	6+A	-	-	-	-	-	-	2.794	40.53	0.00466
1 x 3	M	A	6	-	-	-	-	-	2.566	29.42	0.00428
1 x 4	M	A	3	3	-	-	-	-	1.923	29.45	0.00321
2 x 2	M	A	3	3	-	-	-	-	1.872	30.25	0.00312
2 x 3	M	A	1	1	2	2	-	-	1.673	22.56	0.00279
2 x 4	M	A	1	1	1	1	1	1	1.603	17.66	0.00267

Table 8.3: Results of running the switching scheme over 600 time steps

Partition	No. of Fixed Networks on Processor Number 0	1	2	3	4	5	6	7	Time (s)	Scalability(%)	Min Length of Sampling Period(s)
1 x 1	6+A	-	-	-	-	-	-	-	45.564	100.00	0.00380
1 x 2	M	6+A	-	-	-	-	-	-	57.297	39.76	0.00477
2 x 1	M	6+A	-	-	-	-	-	-	56.751	40.14	0.00473
1 x 3	M	A	6	-	-	-	-	-	51.683	29.39	0.00431
1 x 4	M	A	3	3	-	-	-	-	38.801	29.36	0.00323
2 x 2	M	A	3	3	-	-	-	-	37.973	30.00	0.00316
2 x 3	M	A	1	1	2	2	-	-	34.163	22.23	0.00285
2 x 4	M	A	1	1	1	1	1	1	33.060	17.23	0.00276

Table 8.4: Results of running the switching scheme over 12000 time steps

Partition	No. of Fixed Networks on Processor Number 0	1	2	3	4	5	6	7	Time (s)	Scalability(%)	Min Length of Sampling Period(s)
1 x 1	6+A	-	-	-	-	-	-	-	203.218	100.00	0.00376
1 x 2	M	6+A	-	-	-	-	-	-	232.155	43.77	0.00430
2 x 1	M	6+A	-	-	-	-	-	-	255.029	39.84	0.00472
1 x 3	M	A	6	-	-	-	-	-	230.471	29.39	0.00427
1 x 4	M	A	3	3	-	-	-	-	175.844	28.89	0.00326
2 x 2	M	A	3	3	-	-	-	-	174.567	29.10	0.00323
2 x 3	M	A	1	1	2	2	-	-	163.320	20.74	0.00302
2 x 4	M	A	1	1	1	1	1	1	150.237	16.91	0.00278

Table 8.5: Results of running the switching scheme over 54000 time steps

steps. These times are also more consistent with processor partitions of the same size taking times of an extremely similar length. The actual run times are much slower, due to the inferior transputer processor speed, but any inconsistencies in calculation times would appear to be unique to the PowerPC array. This is probably due to a better balance between processor and communication speeds present on the transputer array.

8.7 Optimisation

The communication strategy has been minimised already. The main optimisation potential lies in code optimisation.

The usefulness of the adaptive component can be significantly increased. As the convergence rate of neural networks is so slow, there is no point concentrating on a real time adaptive component in any sense. However, the potential lies in taking advantage of the fine tuning element, that allows the adaptive component to adapt to the wear and tear of a system (as with more conventional adaptive control systems). In order to take full advantage of this the fixed models themselves must be updated. The best way to achieve this is to update the model that corresponds to J_{cur} from which the adaptive model has been tuning with the current adaptive model weights, before the adaptive model is updated with the model weights that correspond to the cost function J_{new}.

Although the nonlinear switching scheme has been limited to plant environments that can be stabilized by a zero input, the scheme can easily by extended to include observers and stabilisers as described in sections 7.3.1 and 8.2.3 respectively. This will increase the computational intensity and the adaptive component will have to fine tune two more components as a result.

Partition	No. of Fixed Networks on Processor Number 0	1	2	3	4	5	6	7	Time (s)	Scalability(%)	Min Length of Sampling Period(s)
1 x 1	6+A	-	-	-	-	-	-	-	345.004	100.00	0.00383
1 x 2	M	6+A	-	-	-	-	-	-	435.497	39.61	0.00484
2 x 1	M	6+A	-	-	-	-	-	-	438.930	39.30	0.00488
1 x 3	M	A	6	-	-	-	-	-	410.554	28.01	0.00456
1 x 4	M	A	3	3	-	-	-	-	310.056	27.82	0.00345
2 x 2	M	A	3	3	-	-	-	-	282.233	30.56	0.00314
2 x 3	M	A	1	1	2	2	-	-	275.992	20.83	0.00307
2 x 4	M	A	1	1	1	1	1	1	256.130	16.84	0.00285

Table 8.6: Results of running the switching scheme over 90000 time steps

Partition	No. of Fixed Networks on Processor Number 0	1	2	3	4	5	6	7	Comms To Fixed Time (s)	Comms To Adapt Time (s)	Total Time (s)
1 x 1	6+A	-	-	-	-	-	-	-	0.0000	0.0000	0.0000
1 x 2	M	6+A	-	-	-	-	-	-	0.0369	0.0000	0.0369
2 x 1	M	6+A	-	-	-	-	-	-	0.0368	0.0000	0.0368
1 x 3	M	A	6	-	-	-	-	-	0.0467	0.0368	0.0835
3 x 1	M	A	6	-	-	-	-	-	0.0465	0.0368	0.0833
1 x 4	M	A	3	3	-	-	-	-	0.0510	0.0369	0.0879
4 x 1	M	A	3	3	-	-	-	-	0.0509	0.0368	0.0877
2 x 2	M	A	3	3	-	-	-	-	0.0409	0.0368	0.0777
2 x 3	M	A	1	1	2	2	-	-	0.0470	0.0368	0.0838
3 x 2	M	A	1	1	2	2	-	-	0.0470	0.0368	0.0838
2 x 4	M	A	1	1	1	1	1	1	0.0508	0.0368	0.0876
4 x 2	M	A	1	1	1	1	1	1	0.0510	0.0368	0.0878

Table 8.7: Time taken to communicate trained networks to required processors on the transputer array

Partition	No. of Fixed Networks on Processor Number 0	1	2	3	4	5	6	7	Time (s)	Scalability(%)	Min Length of Sampling Period(s)
1 x 1	6+A	-	-	-	-	-	-	-	34.806	100.00	0.0580
1 x 2	M	6+A	-	-	-	-	-	-	34.101	51.03	0.0568
2 x 1	M	6+A	-	-	-	-	-	-	34.102	51.03	0.0568
1 x 3	M	A	6	-	-	-	-	-	33.962	34.16	0.0566
3 x 1	M	A	6	-	-	-	-	-	33.962	34.16	0.0566
1 x 4	M	A	3	3	-	-	-	-	23.410	37.17	0.0390
4 x 1	M	A	3	3	-	-	-	-	23.410	37.17	0.0390
2 x 2	M	A	3	3	-	-	-	-	23.375	37.23	0.0390
2 x 3	M	A	1	1	2	2	-	-	19.913	29.13	0.0332
3 x 2	M	A	1	1	2	2	-	-	19.910	29.14	0.0332
2 x 4	M	A	1	1	1	1	1	1	16.398	26.53	0.0273
4 x 2	M	A	1	1	1	1	1	1	16.395	26.54	0.0273

Table 8.8: Results of running the switching scheme over 600 time steps on the transputer array

8.8 Conclusions

In section 7.3.1, the results in (Levin and Narendra 1996) were tested and verified. Improvements to the training process were suggested and analysed. Although reasonable estimation of the states was demonstrated when training a network from input and output data inputs, better accuracy was obtained using the state vector. Judging by the results shown in (Levin and Narendra 1996), a far harsher constraint on the plant outputs (i.e. a narrower boundary than even the $y \in (-3.5, 3.5)$) might be necessary. Low errors have been achieved in (Levin and Narendra 1996), although problems in the estimation of x_3 are apparent from their results.

In section 8.2.2, the identification networks introduced in (Levin and Narendra 1996) were verified. To increase the accuracy, it was necessary to put constraints on the plant by resetting the plant states and network inputs to zero if the plant output exceeded certain boundaries. It is almost certain that similar constraints were necessary in (Levin and Narendra 1996). The example presented was open loop stable, substantially reducing the computational overheads, since networks NN_Φ and NN_g can be discarded (the networks required for the purposes of stabilisation). Limiting plants to this type also has the effect of simplifying the components of the switching scheme in section 8.3.

In section 8.2.3, the stabilisation problem was considered. An example in (Levin and Narendra 1996) was stabilised with varying degrees of success. A number of related improvements and refinements to aid convergence were presented and discussed. The accuracy of an observer network trained to estimate the system states from input/output data was demonstrated to be sufficient in this case (all networks in the section were trained from model data but tested with observer

estimation) and only slight errors were present - which may be as much to do with solutions to the differential equations that exist around the equilibrium point as with errors in estimation.

There may be problems using systems that require a stabiliser in the nonlinear switching scheme (see section 8.3). The training parameters required to get each stabiliser to work may be unique to each plant environment. However, provided each environment is covered, a resettable adaptive model could still be a solution to this problem. The implementation of observers and stabilisers into the switching scheme was not considered in this thesis.

Following the treatment of the stabilisation problem, a controller network was successfully trained to control a plant trajectory to follow a sinusoidal reference signal. This was achieved without any constraints being placed on the plant, hence justifying the resetting constraints placed on the identifier networks NN_h and NN_d. The controller network converged with no additional training biases required.

In section 8.3, successful control was demonstrated with a six environment plant that changes frequently over a short period of time. The usefulness of the adaptive component only becomes apparent when it is used to fine tune the fixed model components of the system, allowing the plant to be controlled when its operating environments are known at the outset, but also allowing the system to cope with unknown wear and tear of the system over a relatively long period of time (when compared to normal operation time).

A good example of an area to which this is applicable is a robot arm that operates in environments that can not be modelled by a linearised model (or set of models). This offers another level of complexity to the previous linear switching

scheme and would allow a large variation of load to be lifted by the robot arm (when compared to the overall mass of the arm). This in turn would allow the mass of the arm to be reduced in relation to the load masses encountered in run time.

The parallelisation of the scheme has been justified since a significant reduction in the minimum sampling period has been achieved allowing the sampling rate to be increased which in turn increases the accuracy of control.

Chapter 9

General Conclusions and Further Work

9.1 Conclusions

After presenting introductory material in chapters 1 to 3, the first original work appeared in chapter 4 with verification work for systolic architectures introduced in (Li 1990) that were for the parallel computation of difference equations required as part of the control process. As part of this process, parallel verification and analysis techniques were introduced and used to model the architectures from (Li 1990). Such architectures were shown to be highly parallel and scalable assuming efficient coding was employed.

Possible parallel processing schemes for the computation of matrix-vector multiplication, arising in Dynamic Matrix Control, were introduced in chapter 5. Although it is possible to decompose the basic calculation into a number of parallel components, the communication overheads required to transmit the data to the worker processors is comparatively high. Even in problems where message sizes

were at an optimum for communication, only a small speed-up was achieved. For MIMD architectures of the type considered in this work, appreciable speed up from parallel processing would only be achievable if the balance of communications to computations altered significantly in favour of communications.

Chapter 6 introduced the concept of parallel multiple model adaptive control. The controller corresponding to the best model from a large number of fixed models and a real time adaptive model was *switched to* and used to control a linear plant that undergoes large discontinuous changes in its dynamics during normal run time. The inherent algorithm here was highly parallelisable and the level of parallelism was limited to the number of fixed models employed and a minimum granularity (models per processor) below which no further speed-up was possible. The effectiveness of such systems depends on the maximum number of fixed models applicable to a given plant. This limit is comparatively small (at most a few hundred), making the parallelism most applicable to a small number of low cost processors as opposed to available massively parallel machines.

Chapter 7 provided some essential theory as a foundation for the neural network schemes of chapter 8.

Chapter 8 extended the switching schemes in chapter 6 to nonlinear plants. Extensive research is presented into the application of neural networks to all areas of the control process before a switching scheme is constructed from neural network identifiers and controllers that have been pre-trained. Such systems require the existence of acceptable mathematical models of each environment that the plant might occupy during any run time. Neural networks are trained to model the plant trajectories from past input and output data - with a neural network trained to each known environment. This allows the overall control system to cope with large discontinuous changes in dynamics during normal run time, while having

the ability to fine tune the neural networks during run time.

All parallel solutions in this thesis have been developed around the assumption that only a small number of low cost processors are available. The solutions have largely been for problems where the ratio between computation and communication has been high - a key property of problems that are highly parallelisable - with the exception of chapter 5 where only a small amount of speed-up was achievable (and then only in optimal conditions) as a result.

The switching schemes of chapters 6 and 8 offer great promise. Frequently, the practical solution to plants which are not time invariant is to make the dynamic changes that the plant undergoes negligible. Robot arms are of a much larger mass than their loads for this reason. If the constraint on mass is removed from the system with the implementation of a multiple model switching scheme, the arms can be made much lighter - making them cheaper and having important implications in the space industry where payloads are critical.

The adaptive component of the nonlinear switching scheme may at first seem obsolete. However, if the control system is for a system used often, the adaptive component can fine tune models to slow changes in plant dynamics such as that caused by ageing components and other wear and tear of the system through frequent use.

The thesis has successfully identified a number of techniques in control system engineering in which parallel processing can be successfully employed inexpensively. It has also shown that not all inherently parallel algorithms are suitable for parallelisation using commodity processors. Through the implementation of the switching schemes for linear and nonlinear plants steps have been made towards widening the field of adaptive control beyond slow-time varying plants and in the

case of nonlinear plants, suggesting a generalised control scheme. As to how useful parallel processing will be in the future is a matter of debate. Over the last five years little has been suggested in the field with fastness of control coming from the implementation of algorithms on increasingly fast sequential machines. The next five years might see parallel control schemes die out completely. With some plants, modern digital control has already reached a near optimal level and parallel processing offers nothing more there. However, with a plant whose dynamics can change in a large discontinuous step, a sampling period as short as possible and as short a delay in the generation of the control input will always be two important goals. As sequential processing has an upper theoretical limit of not much more than the gigahertz processors on the horizon, the only way of increasing speed beyond that limit lies in the connecting together of very fast sequential processors by a fast ethernet link (or any link that allows fast communication between processors). The future will perhaps see more application specific computers tuned to specialist calculations - of which one field that could benefit is adaptive and self-tuning control.

9.2 Further Work

Several areas of further interest immediately arise from the work presented in this thesis. Some suggestions for further work are now presented.

1. Update the systolic architectures of chapter 4 to include modern processors.
2. The implementation of systolic architectures to the control input generation and Recursive Least Squares component calculations of the switching scheme in chapter 6 - allowing more complex systems models to be constructed.
3. The extension of the multiple model switching scheme in chapter 6 to linear MIMO systems.
4. The implementation of the nonlinear multiple model switching schemes of chapter 8 to systems that are not open-loop stable with the introduction of neural network observers and stabilisers to the scheme.
5. An investigation into the specific design implications of extending adaptive control to time invarient systems - for example by allowing the mass of robot arms to be reduced.
6. An investigation into the industrial applications of such systems.
7. Further investigation into areas of control that could benefit from parallelisation with the aim of producing a generic parallel toolbox for control systems design and implementation.
8. An investigation into the application of different neural network models to the switching scheme of chapter 8 (for example, B-Splines, neuro-fuzzy models, networks that introduce transfer function models as node functions).

Bibliography

Aseltine, J. A, A. R Mancini, and C. W Sarture, 1958 (December). A survey of adaptive control systems. *IRE Transactions on Automatic Control* **3**, 102—108.

Aström, K. J, 1983 (September). Theory and applications of adaptive control - a survey. *Automatica* **19**, 471—486.

Autenreith, T, 1996. An investigation into adaptation and learning based on multiple models switching and tuning. Technical report, The University of Southampton.

Bakkers, A (Ed.), 1989. *Applying Transputer Based Parallel Machines.* IOC.

Baude, F, 1989. *Topologies for Large Transputer Networks: Theoretical Aspects and Experimental Approach.* In (Bakkers 1989).

Baxter, M, M. O Tokhi, and P. J Fleming, 1994. Parallelising algorithms to exploit heterogeneous architectures for real-time control systems. In *IEE Proceedings of Control '94 21-24 March 1994. Conference publication number 389*, pp. 1266—1271.

Brown, D, Y Li, E Rogers, and O. R Tutty, 1995. Verification of parallel architecture for real time feedback controllers. In *Proceedings of 3rd IFAC/IFIP Workshop on Algorithms and Architectures for Real-Time Control (AARTC '95), Ostend, Belgium, 31st May - 2nd June 1995*, pp. 355—360.

Brown, D, O. R Tutty, and E Rogers, 1996. Computer aided design and evaluation of multiple model adaptive control schemes. In *Proceedings of 7th Symposium on Computer Aided Control Systems (CACSD ' 97), Gent, Belgium, 28th - 30th April 1997*, pp. 85—90.

Brown, D, O. R Tutty, and E Rogers, 2000. Parallel implementation of dynamic matrix control algorithms. *International Journal of Control.* To be published.

Brown, M and C. J Harris, 1993. *The B-spline Neurocontroller*, pp. 134—167. In *Parallel Processing in a Control Systems Environment* (Rogers and Li 1993).

Chen, Y. C and C. C Teng, 1995. A model reference control structure using a fuzzy neural network. *Fuzzy Sets and Systems* **73**, 291-312.

Chien, I. L, 1996 (September). Simple empirical nonlinear model for temperature-based high-purity distillation columns. *Aiche Journal 42*(9), 2692—2697.

Chisci, L and G Zappa, 1993. *Systolic Architecture for Adaptive Control*, pp. 36—71. In (Rogers and Li 1993).

Cybenko, G, 1989. Approximation by superposition of sigmoidal function. *Mathematics of Control Signals and Systems 2*(4), 303—314.

Dowd, K, 1993. *High Performance Computing*. O'Reilly and Associates Inc.

Elliott, H, R Cristi, and M Das, 1985 (April). Global stability of adaptive pole placement. *IEEE Transaction on Automatic Control 30*(4), 348—356.

Fleming, P. J (Ed.), 1988. *Parallel Processing in Control: the transputer and other architectures*. IEE Control Engineering Series 38. Peter Peregrinus Ltd.

Föllinger, O, 1993a. *Nichtlineare Regelungen I.* Müchen: Oldenburg Verlag. (in German).

Föllinger, O, 1993b. *Nichtlineare Regelungen II.* Müchen: Oldenburg Verlag. (in German).

Franklin, G. F and J. D Powell, 1980. *Digital Control of Dynamic Systems*. Addison Wesley.

French, M and E Rogers, 1997. Approximate parameterisations for adaptive feedback linearisation. In *Proceedings of the 1997 IEEE conference on decision and control*, pp. 4665—4670.

French, M and E Rogers, 1998. Input/output stability theory for direct neuro-fuzzy controllers. *IEEE Transactions on fuzzy systems* **6**, 331—345.

French, M, C Szepesvari, and E Rogers, 2000. Uncertainty, performance and model dependancy in approximate adaptive nonlinear control. *IEEE Transactions on Automatic Control* **45**. To appear May 2000.

Gaston, F. M. F, J Kadlec, and J Schier, 1994. The block regularised linear quadratic optimal controller. *I.E.E Proceedings of Control 94*, 1254—1259. 21 - 24 March, Conference publication No 389.

Genceli, H and M Nikalaou, 1995 (September). Design of robust constrained model predictive controllers with volterra series. *Aiche Journal 41*(9), 2098—2107.

Goodwin, G. C and D. Q Mayne, 1987. A parameter estimation perspective of continuous time model reference adaptive control. *Automatica 23*(1), 57—70.

Goodwin, G. C and K. S Sin, 1984. *Adaptive filtering prediction and control.* Information and systems science series. Englewood cliffs, New York: Prentice Hall.

Goulding, J. R, 1991. Adaptive transfer functions. In *Proceedings of International Joint Conference on Neural Networks 1991*, Volume 2.

Harris, C. J and S. A Billings (Eds.), 1985. *Self-tuning and Adaptive Control: Theory and Applications* (2nd ed.). London, UK: Peter Peregrinus Ltd.

Hoare, C. A. R, 1978 (August). Communicating sequential processes. *Communications of the ACM 21*(8), 666—677.

Hopfield, J, 1982. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences* **79**, 2554—2558.

Hornik, K, M Stinchcomb, and H White, 1989. Multilayer feedforward networks are universal approximators. *Neural Networks* **2**, 359—366.

Hunt, D. J, 1989 (April). Amt dap - a processor array in a workstation environment. *Computer Systens Science and Engineering 4*(2), 107—114.

Isidori, A, 1989. *Nonlinear control systems: an introduction* (2nd ed.). New York: Springer-Verlag.

Jacobs, O. L. R, 1964 (May). Two uses of the term 'adaptive' in automatic control. *IEEE Transactions on Automatic Control* **9**, 574—575.

Kalman, R. E, 1958. Design of a self-optimizing control system. *IRE Transactions on Automatic Control 4*(1), 65—68.

Kreisselmeier, G and R Lozano, 1996 (November). Adaptive control of continuous-time overmodeled plants. *IEEE Transactions on Automatic Control 41*(12), 1779—1794.

Kröse, B and P Van der Smagt, 1996. *An introduction to Neural Networks* (8 ed.). Kruislaan 403, NL-1098 SJ Amsterdam: The University of Amsterdam.

The entire text can be downloaded from the Faculty of Mathematics and Computer Science's homepage.

Kung, S. Y, 1988. *VLSI Array Processors.* Information and System Sciences. Prentice Hall.

Kwan, H. K, 1987 (December). Systolic realization of linear phase fir digital filters. *I.E.E.E Transactions on Circuits and Systems 34*(12), 1604—1605.

Lang, S, 1983. *Real Analysis.* New York: Addison-Wesley.

Lee, J, M Morari, and C Garcia, 1994. State-space interpretation of model predictive control. *Automatica 30*(4), 707—717.

Leighton., F. T, 1992. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers Inc.

Levin, A. U and K. S Narendra, 1993 (March). Control of nonlinear dynamical systems using neural networks: Controllability and stabilization. *IEEE Transactions on Neural Networks 4*(2), 192—205.

Levin, A. U and K. S Narendra, 1996 (January). Control of nonlinear dynamical systems using neural networks - part ii: Observability, identification, and control. *IEEE Transactions on Neural Networks 7*(1), 30—42.

Levin, M. J, 1958. Methods for the realization of self-optimizing systems. In *Proceedings of the ASME-IRD Conference, Newark, Del., April 2-4 1958.* ISA Paper No. FCS 2-58.

Li, Y, 1990 (July). *Concurrent Architectures for Real-Time Control.* Ph. D. thesis.

Li, Y and E Rogers, 1993. *IMS A100 / Transputer Based Heterogeneous Architectures for Embedded Control Problems.* In (Rogers and Li 1993).

Lin, H, 1986. New vlsi systolic array design for real-time digital signal processing. *IEEE Transactions on Circuit and Systems* **33**, 673—676.

Ljung, L, 1977. Analysis of recursive stochastic algorithms. *Automatic Control* **22**, 551—575.

Ljung, L, 1999. *System Identification: Theory for the User* (2nd ed.). Information and System Sciences. Englewood Cliffs, New Jersey 07632: P. T. R Prentice Hall.

Ljung, L and T Söderström, 1983. *Theory and Practice of Recursive Identification.* Cambridge, MA: MIT Press.

May, D and R Shepherd, 1990. Occam and the transputer. *Lecture Notes in Computer Science* **424**, 329—353.

McWhirter, J. G and I. K Proudler, 1994 (March). A systolic array for recursive least squares estimation by inverse updates. In *Proceedings of Control '94, 21-24 March 1994, conference publication No 389*, pp. 1272—1277.

Middleton, R. H, G. C Goodwin, D Hill, and D. Q Mayne, 1988. Design issues in adaptive control. *IEEE Transactions on Automatic Control 33*(1), 50—57.

Morse, A, D Mayne, and G Goodwin, 1992 (September). Applications of hysteresis switching in para neter adaptive control". *IEEE Transactions on Automatic Control 37*(9), 1343—1354.

Narendra, K and J Balakrishnan, 1997 (February). Adaptive control using multiple models. *IEEE Transactions on Automatic Control 42*(2), 171—187.

Narendra, K, J Balakrishnan, and M Ciliz, 1995 (June). Adaptation and learning using multiple models, switching and tuning. *IEEE Control Systems Magazine*, 37—50.

Narendra, K. S and K Parthasarathy, 1991 (March). Gradient methods for the optimization of dynamical systems containing neural networks. *IEEE Transactions on Neural Networks* **2**, 252—261.

Parsytec, 1993. *Software Documentation and Manual Pages. Parix version 1.2.*

Parsytec, 1994 (May). *Parsytec PowerXplorer hardware user guide. Revision 1.1.*

Rogers, E and Y Li (Eds.), 1993. *Parallel Processing in a Control Systems Environment.* Systems and Control Engineering. London: Prentice Hall International(UK) ltd.

Rumelhart, D. E, G. E Hinton, and R. J Williams, 1986. *Learning internal representations by error propagation*, Volume 1, Chapter 8, pp. 318—362. Cambridge, MA: MIT Press.

Sarimveis, H, H Genceli, and M Nikalaou, 1996 (September). Design of robust nonsquare constrained model predictive control. *Aiche Journal 42*(9), 2582—2593.

Sastry, S and M Bodson, 1989. *Adaptive Control. Stability, Convergence and Robustness.* Prentice Hall International Inc.

Slotine, J.-J. E and W Li, 1991. *Applied Nonlinear Control.* Englewood Cliffs: Prentice-Hall.

Sontag, E. D, 1990. *Mathematical Control Theory.* New York: Springer-Verlag.

Takeda, K, O Tutty, and D Nicole, 1999. Parallel discrete vortex methods on commodity supercomputers; an investigation into bluff body far wake behaviour. In *Proceedings of the third international workshop on vortex flows and related numerical methods*, Volume 7, pp. 418—428.

Trew, A and G Wilson (Eds.), 1991. *A Survey of Available Parallel Computing Systems.* London: Springer-Verlag.

Tully, A and M Surridge, 1993 (October). Generic parallel software components and techniques for simulation and control. Technical report, The Parallel Applications Centre, The University of Southampton.

Weller, S and G Goodwin, 1994 (July). Hysteresis switching adaptive control of linear multivariable systems. *IEEE Transactions on Automatic Control 39*(7), 1360—1375.

Wismer, D. A and C. H Wells, 1972. A modern approach to industrial process control. *Automatica* **8**, 117—125.