CRANFIELD UNIVERSITY

COLLEGE OF AERONAUTICS

Department of Aerospace Science

MPhil THESIS

Academic Year 1993-4

A. COOPER

The Application of Neural Networks to Spacecraft Control

Supervisor : D.J. Lewis

August 1994

Acknowledgements

I would like to thank George Game of British Aerospace Space Systems Limited, Earth Observation and Science Division, Filton, Bristol, for taking the time to meet me at the beginning of the year, and for providing me with initial guidance with this work.

I would also like to acknowledge the help of my supervisor, John Lewis, for giving me general guidance and support during the course of the year.

Abstract

This thesis investigates how two neural network-based control techniques can be applied to a specific spacecraft control problem.

The neural networks used are simple backpropagation networks, consisting of one or more tansigmoidal neurons (neurons with tanh transfer functions) in a hidden layer, and a linear neuron in the output layer. The neural network control techniques investigated here are Direct Model Inversion and Indirect Model Inversion.

The spacecraft control problem is that of reducing the vibrations of a spacecraft payload. The source of the vibrations is a mass imbalance in one of the reaction wheels of the spacecraft. Four components are represented in the spacecraft model. These are rigid body inertia, solar array flexure, fuel slosh and payload vibration. A simple sinusoidal signal is used to model the disturbance torque produced by the reaction wheel mass imbalance. The complete model is broadly based on the Solar Heliospheric Observatory (SOHO) that is due for launch in 1995.

Each of the neural network control techniques used is shown to be successful in reducing the effects of the disturbance torques on the spacecraft payload. However, in each case, a simple positional feedback gain term provides more effective and reliable control.

Contents

Figures

1 Introduction

In the last few years interest in the field of neural networks has increased considerably. Although the first work that was carried out in the field dates back to the early 1940's [1], it is only recently that most neural network development work has occurred. This is due in part to certain breakthroughs that have recently occurred in the field, but also partly because of the considerable increase in computational power that has recently become available to researchers.

Neural networks consist of a collection of simple processing units referred to as neurons, which are generally arranged in layers. The neurons in one layer interact with those in the next layer by passing their outputs along weighted connections. The weights of each of the connections in the network can be updated during operation by using a particular algorithm that calculates appropriate weight changes in accordance with the performance of the network.

Knowledge outside the field of neural networks of the existence of neural network technology has also increased significantly in recent years. Many people are currently aware of the existence of neural networks but often the technology is not well understood. This is probably due to the fact that neural networks are based, albeit very loosely, on the biological function of the human brain. Not surprisingly, people often see current neural network applications as an attempt to imitate the function of the brain in some fashion. However, the reality is that most of the neural network applications that are currently in existence are no more than simple computational algorithms. Although these algorithms are based on a collection of individual units referred to as neurons, often no more than say 50 of such units are incorporated in any network. This is a far cry from the biological structure of the human brain where some 5×10^9 units known as neurons are believed to be in existence.

Another reason for likening the function of neural networks to the function of the human brain is that they can be used to "learn" how to perform tasks. However, the term "learn" really means no more than a neural network adapting itself according to a rigid computational algorithm, that calculates appropriate connection weight updates from some measure of the performance of the network.

Despite the loose connection between current neural networks and the function of the human brain, a number of neural network applications have been shown to perform well on the sort of "fuzzy" problems which humans find easy to solve but for which traditional computational approaches have often been left floundering. A prime example of this is in the area of pattern recognition, where traditional computing techniques have often struggled, but where some simple neural network techniques have provided powerful solutions.

Similarly, there are a number of problems relating to current or future space missions that cannot be easily solved by conventional computational techniques, and that may be suited to the future application of neural network based techniques. These include, for example:

(i) On-board fault detection, monitoring and reconfiguration: A neural network could be used recognise the characteristic signature of the failure of a particular spacecraft component. This information could then be used by an on-board fault tolerant system or by ground staff to decide upon an appropriate operational strategy.

(ii) Rendezvous and docking problems: A neural network could possibly be used here as the basis of a guidance algorithm for use in rendezvous and docking of spacecraft to a future space station.

(iii) Remote landing site selection: Various space missions have been envisaged that would require a spacecraft to initially map a remote body and to then choose a suitable landing site autonomously. It is possible that the pattern recognition abilities of neural networks would be well suited to the task of assessing the suitability of such landing sites.

This simple discussion does not by any means represent an exhaustive analysis of the possible space applications of neural networks. The discussion is simply provided to give a flavour of possible future developments.

In this thesis, however, it is the control potential of neural networks that is investigated. The main basis for using neural networks for control purposes is that theoretical proofs exist [2-5] that show that certain types of neural network can approximate any reasonable function (linear or non-linear) to any desired level of accuracy. This then

gives the hope that a neural network could be trained to approximate the function that maps plant outputs to desired control signal inputs for a plant.

The spacecraft control problem that is considered in the work presented here is based on work already published by other workers [6]. The problem considered is one of spacecraft payload vibrations induced by a reaction wheel mass imbalance. The dynamical model of the spacecraft that is used in this work is loosely based on the dynamics of the Solar Heliospheric Observatory (SOHO) spacecraft that is due for launch in 1995. The model incorporates four components: rigid body inertia, solar array flexure, liquid fuel slosh and payload vibration. The model is represented as a series of connected transfer functions using the Simulink control system simulation software that is available with the Matlab mathematical analysis program.

Two neural network control methods are considered in this work. These are commonly referred to in the literature as Direct Model Inversion and Indirect Model Inversion. The neural networks that are required in each of these methods are constructed using the Neural Network Toolbox that is also available with the Matlab package.

In the discussion of the work that follows, the following layout has been adopted: In chapter 2, an introduction to the field of neural networks is given. This has been provided so that the work carried out in this thesis can be put into context by a reader with little or no experience of neural network technology. Chapter 3 discusses the two main methods that have been proposed for using neural networks for control purposes. Although a large number of neural network control methods have been suggested, each method is often just a slight variation on one of the two main methods discussed here. In chapter 4, a step by step discussion of how the spacecraft model that was used for the work in this thesis was constructed. In chapters 5 and 6, a discussion of the implementation of the two neural network control methods described in chapter 3 is given along with the results obtained using each of the methods. Finally in chapter 7, a summary of the results that were obtained is given along with some conclusions and some suggestions of areas in which further study might be considered.

2 Introduction to Neural Networks

Over the last few years, the subject of neural networks has grown rapidly. However, it is still a relatively new technology, and the subject still remains in its infancy. The subject is unusual in that its areas of application span a broad spectrum of disciplines. For these reasons, no prior knowledge of neural networks is assumed here, and a concise introduction to the subject is presented in this chapter.

A neural network consists of a collection of simple units called neurons that are connected to each other in some particular way. The function of a neuron is to receive information, process it in some simple way, and then pass the resulting information on to the other neurons in the network to which it is connected, subject to the strengths of each of these connections. Although a single neuron is not capable of much by itself, a collection of them connected in a network can have powerful capabilities.

Neural networks can be implemented in hardware, for fast fault-tolerant operation. However, both development work and final implementation are often carried out using software simulations. There are many computer programs commercially available that can be used to run neural network simulations.

During the course of this work, the Neural Network Toolbox which is available with the Matlab mathematical analysis software was used. In the introduction that follows, the notation that is adopted is similar to that used in the Matlab Neural Net Toolbox [7].

2.1 Neural Network Architecture

The term "architecture" in neural network technology refers to the way in which the individual neurons in a network are connected together, how they pass information between each other, and the nature of the basic processing that they each carry out. The simplest neural network architecture is that of a single neuron.

2.1.1 Model of a Single Neuron

A single neuron with multiple inputs can be modelled as shown in figure 2.1.

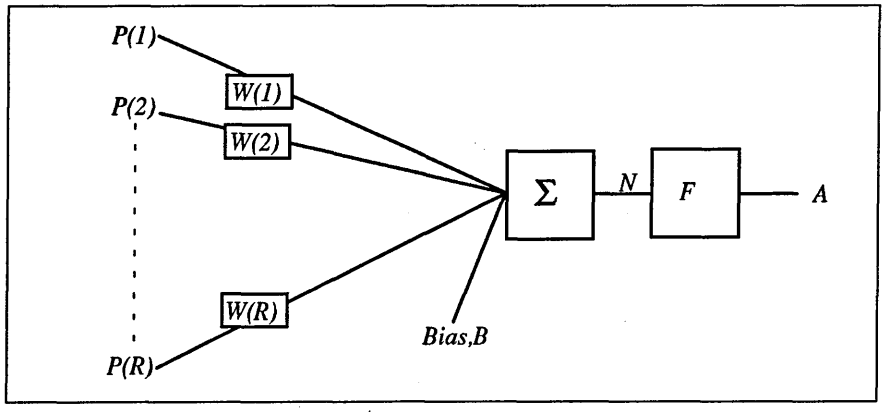

Fig. 2.1 : Model of a single neuron

In figure 2.1, the following nomenclature is used:

P(1), P(2)...P(R)	are the *R* inputs to the neuron
W(1), W(2)...W(R)	are the weights to be applied to each of the inputs to the neuron
B	is a bias
Σ	is a summer
F	is some transfer function (see later)
N	is the input to the transfer function
A	is the output of the neuron

The input *N* to the transfer function in the simple neuron model is given by

$$N = W(1)P(1) + W(2)P(2) + \ldots\ldots + W(R)P(R) + B \tag{2.1}$$

If the *R* inputs to the neuron are stored in a column vector *P* given by

$$P = \begin{pmatrix} P(1) \\ P(2) \\ P(R) \end{pmatrix} \tag{2.2}$$

and the weights that are to be applied to the R inputs are stored in a matrix W given by

$$W = \begin{pmatrix} W(1) & W(2) & W(R) \end{pmatrix} \tag{2.3}$$

then the input, N, to the transfer function is given by

$$N = W \times P + B \tag{2.4}$$

and the output of the single neuron can be written as

$$A = F(W \times P + B) \tag{2.5}$$

2.1.2 A Single Layer of Neurons

The approach adopted above is easily extended to a layer of several neurons, each with several inputs. Figure 2.2 gives a schematic representation of a layer of S neurons, each with R inputs.

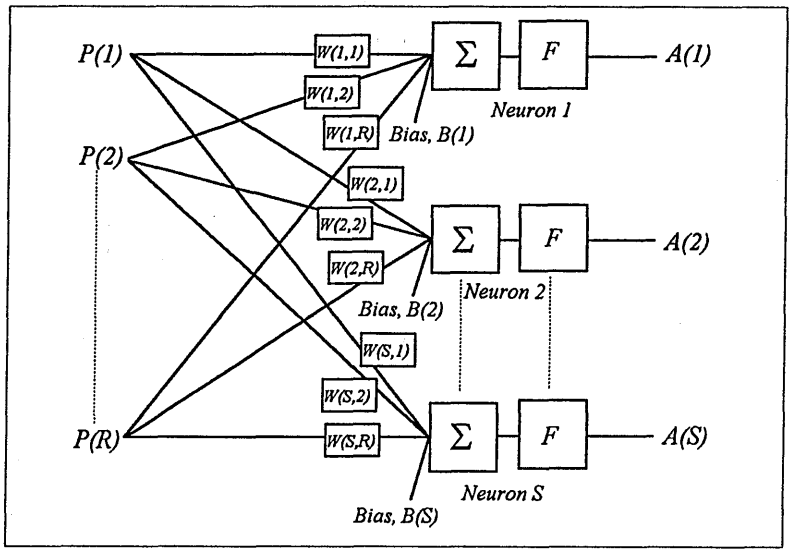

Fig. 2.2 : A single layer of neurons

As before, there are R inputs stored in the input matrix P. However, there are now S neurons and therefore $S \times R$ weights and S outputs.

The weight matrix W now becomes:

$$W = \begin{pmatrix} W(1,1) & W(1,2) & \cdots & W(1,R) \\ W(2,1) & W(2,2) & \cdots & W(2,R) \\ \vdots & \vdots & \vdots & \vdots \\ W(S,1) & W(S,2) & \cdots & W(S,R) \end{pmatrix} \tag{2.6}$$

where $W(x,y)$ is the weight of the y^{th} input to be applied to the x^{th} neuron.

As there are now S biases and S outputs, a bias matrix, B, and an output matrix, A, can also be introduced which are given by:

$$B = \begin{pmatrix} B(1) \\ B(2) \\ \vdots \\ B(S) \end{pmatrix} \quad \text{and} \quad A = \begin{pmatrix} A(1) \\ A(2) \\ \vdots \\ A(S) \end{pmatrix} \tag{2.7}$$

As the inputs, weights and biases are being stored in matrices, the equation for the output of the network, A, can still be used, i.e.,

$$A = F(W \times P + B) \tag{2.8}$$

It can be seen from this that matrix algebra is ideally suited to the task of neural network construction and manipulation, as the same matrix equations can be adopted for networks of different architectures.

2.1.3 Multiple Layers of Neurons

The approach discussed above can now be extended further to a network with several layers of neurons. Figure 2.3 shows one such possible network configuration.

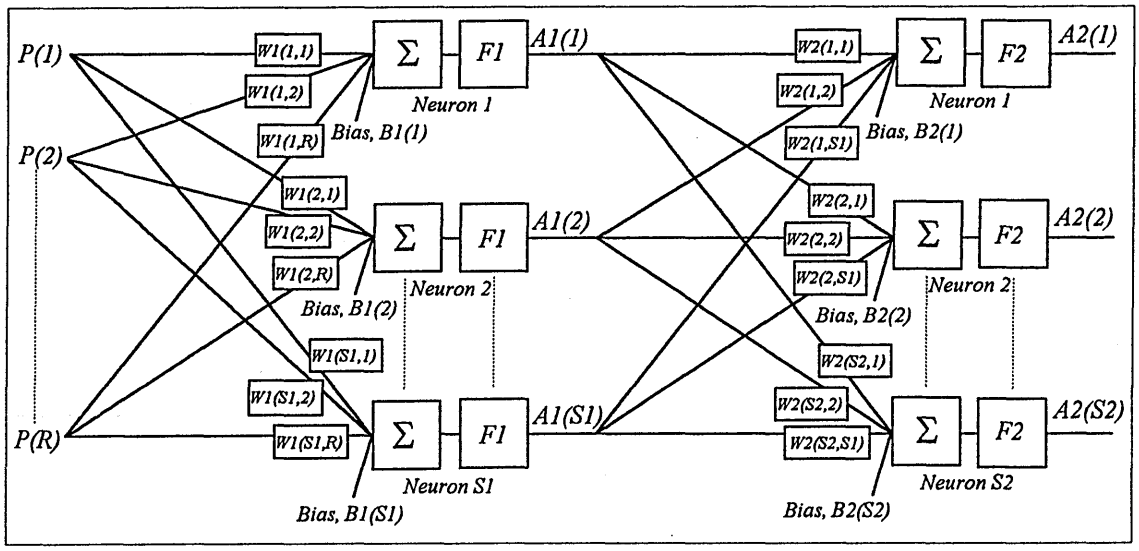

Fig. 2.3 : Neural network with 2 layers

The following points should be noted in figure 2.3:

(i) $S1$ and $S2$ are the number of neurons in layers 1 and 2. There are R inputs to the $S1$ neurons in layer 1, and $S1$ inputs to the $S2$ neurons in layer 2.

(ii) $A1$ is a matrix representing the output of layer 1, and is therefore also the input matrix to layer 2.

(iii) $F1$ and $F2$ are the transfer functions adopted for layers 1 and 2.

The output of the first layer, $A1$, is given by

$$A1 = F1(W1 \times P + B1) \tag{2.9}$$

and therefore the output of the network, $A2$, is given by

$$A2 = F2(W2 \times A1 + B2) \tag{2.10}$$

which gives

$$A2 = F2(W2 \times (F1(W1 \times P + B1)) + B2) \tag{2.11}$$

2.1.4 Batched Inputs

Another important point to note is that inputs to a neural network can be batched. If there are Q different sets of network inputs, these may be batched together by using an input matrix, P, which has Q input vectors, e.g. :

$$P = \begin{pmatrix} P(1,1) & P(1,2) & \cdots & P(1,Q) \\ P(2,1) & P(2,2) & \cdots & P(2,Q) \\ \vdots & \vdots & \vdots & \vdots \\ P(R,1) & P(R,2) & \cdots & P(R,Q) \end{pmatrix} \tag{2.12}$$

The output matrix A that represents the batched outputs will then be of the form

$$A = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,Q) \\ A(2,1) & A(2,2) & \cdots & A(2,Q) \\ \vdots & \vdots & \vdots & \vdots \\ A(R,1) & A(R,2) & \cdots & A(R,Q) \end{pmatrix} \tag{2.13}$$

This feature of being able to batch the inputs to a network is particularly helpful when training a network off-line. In these instances, it is often required to train a network to minimise its error for all inputs rather than for a particular input at a particular instant in time.

2.1.5 Transfer Functions

As yet, no consideration has been given to what form the neuron transfer functions should take. The type of function used will vary from one case to another, depending on the problem to be solved and hence the network architecture adopted. However, one of the following types of function is generally used:

Hard Limit

The Hard Limit transfer function maps neuron inputs in the region $-\infty$ to $+\infty$ into the region 0 to +1. When used with a bias, the Hard Limit transfer function can be used to force a neuron to output a +1 if its input exceeds a certain threshold, but to output a zero for all other inputs. Neurons using this transfer function are particularly suited to problems where a classification or decision has to be made.

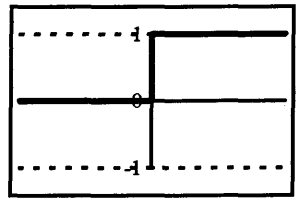

Fig. 2.4 : Hard Limit transfer function

Symmetric Hard Limit

The Symmetric Hard Limit transfer function is similar to the Hard Limit transfer function, although in this case, neuron inputs in the region $-\infty$ to $+\infty$ are mapped into the region -1 to +1. The areas of application for this transfer function are similar to those of the Hard Limit transfer function.

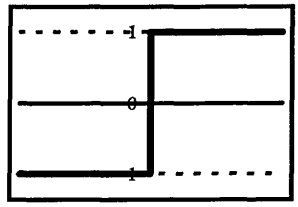

Fig. 2.5 : Symmetric Hard Limit transfer function

Linear

The Linear transfer function is the simplest transfer function used by neurons. The function simply serves to pass the input of the neuron over to its output, adding a bias in the process if one is present. Linear neurons are often used in the output layer of a network, and, via their biases, serve to scale up the outputs from a hidden layer from the region -1 to +1 to the region $-\infty$ to $+\infty$.

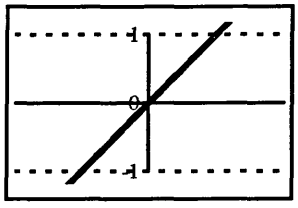

Fig. 2.6 : Linear transfer function

Saturated Linear

The Saturated Linear transfer function gives outputs of -1 and +1 for inputs <1 and >1 respectively. However, for inputs in the region -1 to +1, the function behaves exactly as the Linear transfer function, simply passing its input to its output.

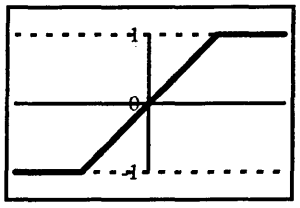

Fig. 2.7 : Saturated Linear transfer function

Log-Sigmoid

The Log-Sigmoid (or logistic sigmoid) transfer function normalizes neuron inputs in the region $-\infty$ to $+\infty$ into the region 0 to +1. The fact that the function is smooth and therefore differentiable means that it is suitable for use in backpropagation networks, where derivatives of the network error are passed back through the network layers.

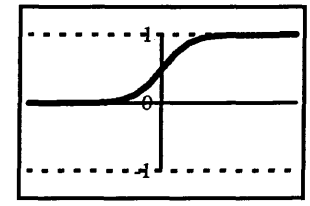

Fig. 2.8 : Log-Sigmoid transfer function

Tan-Sigmoid

The Tan-Sigmoid (or hyperbolic tangent sigmoid) transfer function is similar in function to the log-sigmoid transfer function only in this case neuron inputs in the range $-\infty$ to $+\infty$ are mapped into the region -1 to +1. Again the function is differentiable so it is suitable for use in backpropagation networks.

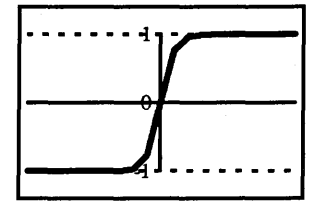

Fig. 2.9 : Tan-Sigmoid transfer function

Gaussian

The Gaussian transfer function is most commonly used in Radial Basis Function (RBF) networks [8-12]. Networks of this type are very powerful as they are guaranteed to only have a single minimum in the network error surface. However, RBF networks are not supported by Matlab v4 which was used for the work carried out here.

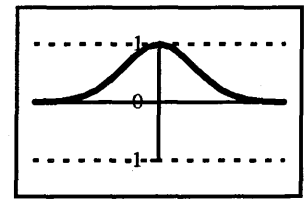

Fig. 2.10 : Gaussian transfer function

2.2 Neural Network Learning

One of the main features of neural networks that distinguishes them from traditional methods of computation is their ability to learn to behave in a certain way in response to a particular input. But how does a neural network learn? The answer is that a neural network learns by changing the weights of the connections between each of its neurons. These weight adjustments are carried out according to some specified rules that define how the weights are to be adjusted in the light of the accuracy of responses to inputs. Although learning can occur whilst the neural network is actually in operation, an off-line training procedure is usually adopted.

2.2.1 Training Procedure

Training a neural network generally involves taking the following steps:

(i) Firstly a training data set is presented to the network. The training data comprises a range of inputs that the network is likely to have when in operation together with corresponding desired outputs.

(ii) Next, the network outputs are compared with the desired outputs, i.e., a measurement is made of the network error.

(iii) Finally some form of learning rule is applied to the network. The function of the learning rule is to adjust the weights of the connections of the network in such a way that the difference between the desired outputs and the outputs achieved (the errors) are in some way minimised.

The above steps are generally repeated over and over until either the measure of the network error falls to a level that is considered acceptable, or until a specified number of training iterations (epochs) have been reached.

2.2.2 Error Surfaces

In order to properly understand the process of learning in most types of neural network, and to be aware of the possible pitfalls in the learning process, it is important to be able to visualise the geometric interpretation of the particular learning rule being used. This can be done by considering a multi-dimensional "weight space" that has one axis for every connection weight in the network, and one additional axis that should be visualised as a "height" that represents the network error. An example of a neural network error surface is shown in figure 2.11 along with a corresponding contour plot for the error surface.

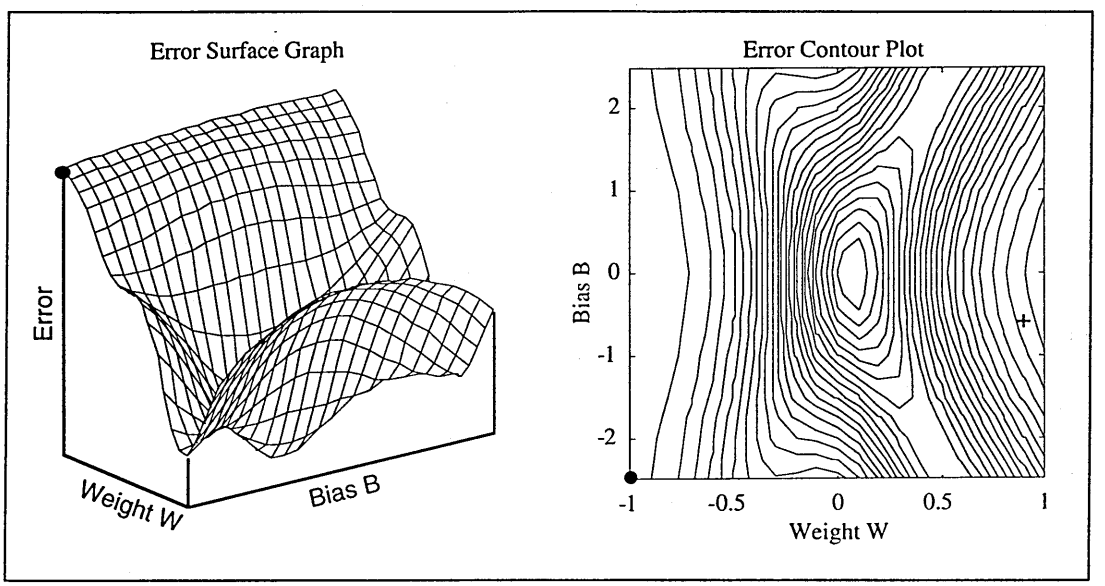

Figure 2.11 : Example of a neural network error surface

For every possible combination of connection weights, there will be a corresponding network error which can be represented by the height of a point in "weight-space". These points form a surface which is known as the "error surface." Thus for a learning rule to reduce the error of a network, it must adjust the weights of the network in such a way that an overall downward path is followed on the multi-dimensional error surface.

2.3 Some Basic Neural Network Types

A particular type of neural network can be characterised by the following choice of parameters:

(i) Type of network architecture (generally number of neurons per layer and number of layers)

(ii) Type of transfer function used in each layer of neurons

(iii) Type of learning rule used in training

The choice of parameters will depend on the nature of the problem to be solved.

So far, a framework has been developed within which the performance of a neural network with an arbitrary choice of inputs, layers, neurons per layer, transfer function and learning rule can be analysed. Some of the popular types of neural networks together with their abilities and limitations are now considered.

2.3.1 The Perceptron

The Perceptron [13-15] is one of the simplest examples of a neural network. It comprises a single layer of neurons with hard limit transfer functions. During training, the outputs of the Perceptron, A, are compared with target outputs, T, and the weights, W, and the biases, B, adapted according to the learning rules :

$$W(i,j)_{new} = W(i,j)_{old} + [T(i) - A(i)] \times P(j) \tag{2.14}$$

and

$$B(i)_{new} = B(i)_{old} + [T(i) - A(i)] \tag{2.15}$$

for all i and j.

Alternatively, these learning rules can be written in matrix form:

$$W = W + EP^T \tag{2.16}$$

and

$$B = B + E \tag{2.17}$$

where E is the error vector (i.e., $E = T - A$).

As Perceptrons use hard limit transfer functions, their outputs can only take on the values one and zero. This means that Perceptrons are generally limited in application to problems involving classification (e.g. pattern classification).

The basic Perceptron model has an important limitation [15]. This is that a Perceptron can only classify linearly separable sets of vectors (that is, vectors that can be separated by a multi-dimensional plane in "weight-space"). For example the boolean exclusive OR (XOR) function cannot be computed by a Perceptron.

2.3.2 Widrow-Hoff Networks

Widrow-Hoff networks [16-19] are similar to Perceptrons but differ in that they have linear transfer functions. In view of their use of linear transfer functions, Widrow-Hoff networks are also known as ADALINES (Adaptive Linear Elements) or MADALINES (Many Adaptive Linear Elements). Widrow-Hoff networks also differ from Perceptrons in that they employ a type of Least Mean Squares learning rule known as the Widrow-Hoff learning rule. This rule adjusts weights and biases according to the magnitude of the network errors rather than just on their existence. This results in a "gradient descent" path being followed on the error surface of the network.

It should also be noted that since linear transfer functions are being used, there is no advantage in having more than one layer of neurons. This is because any multiple layer network of linear neurons can be replaced by a network with just one layer with suitable weights and biases.

The Widrow-Hoff learning rule may be written as:

$$W(i,j)_{new} = W(i,j)_{old} + lr \times [T(i) - A(i)] \times P(j) \tag{2.18}$$

and

$$B(i)_{new} = B(i)_{old} + lr \times [T(i) - A(i)] \tag{2.19}$$

or in matrix form,

$$W = W + lr \times E \times P^T \tag{2.20}$$

and

$$B = B + lr \times E \tag{2.21}$$

where lr is the learning rate. The time it takes the network to learn (if it manages to learn at all) is particularly sensitive to the choice of learning rate. If the learning rate is too high, the network may continually overshoot a minimum in the error surface during its gradient descent. If the learning rate is too low, the network will take an excessively long time to find a minimum in the error surface. Unfortunately, there is no way of determining a priori what value of the learning rate will provide good results. The learning rate must therefore be chosen on a trial and error basis.

The main limitation of Widrow-Hoff networks is that they can only learn linear relationships (since they use linear transfer functions). Nevertheless, when presented with a particular problem, networks of this type will find the best possible solution, that is, the best linear approximation to the function will be made. This is because the error surface of a linear network is a multi-dimensional parabola, so the gradient descent procedure will find the minimum, as long as a suitable learning rate is adopted.

The limitation of using linear transfer functions can be overcome to a certain extent by using an output layer of one or more tan-sigmoid neurons. This approach has been sucessfully applied to the well known non-linear control problem known as the truck backer-upper problem [19,20].

2.3.3 Backpropagation Networks

Backpropagation networks [21,22] are networks that employ the backpropagation learning rule. This learning rule is similar to the Widrow-Hoff learning rule, but it differs in that it is applicable to multi-layered networks (networks with hidden layers) that include neurons with non-linear (but differentiable) transfer functions. The functions used are usually log-sigmoid or tan-sigmoid functions (see earlier) with linear neurons used in the output layer so that outputs with a magnitude > 1 can be produced.

The basic backpropagation learning rules are given by

$$W(i,j)_{new} = W(i,j)_{old} + lr \times D(i) \times P(j) \tag{2.22}$$

and

$$B(i)_{new} = B(i)_{old} + lr \times D(i) \tag{2.23}$$

where *D(i)* are the derivatives of the error with respect to weight (known as delta vectors) and *lr* is the learning rate. With backpropagation, the delta vectors are passed back through the layers of the network from the output layer to the input layer (hence the term backpropagation) and the weights of each layer adjusted.

As with Widrow-Hoff networks, there is no simple way of selecting a suitable learning rate for the backpropagation learning rule. An arbitrary rate must therefore be chosen. Once again this introduces the problems of too small a learning rate causing excessive training times, and too large a learning rate causing the network to continually overshoot the minimum in the network error surface.

2.3.4 Backpropagation with Momentum

One of the potential problems with simple backpropagation networks is that neurons with non-linear transfer functions are used. This results in the possibility of a backpropagation network having a network error surface with more than one error

minimum. This introduces the potential problem that a backpropagation could settle in a local error minimum which may not represent the best solution available.

This problem may be overcome to a certain extent by introducing a "momentum constant" to the backpropagation learning rules given in (2.22) and (2.23). The purpose of the momentum constant is to drive the network up the other side and out of small minima in the error surface so that the network can ultimately settle in the global error minimum of the surface. A diagram showing the progression of a network towards a global minimum in the error surface is shown in figure 2.12 along with a plot of the associated drop in sum-squared error of the network.

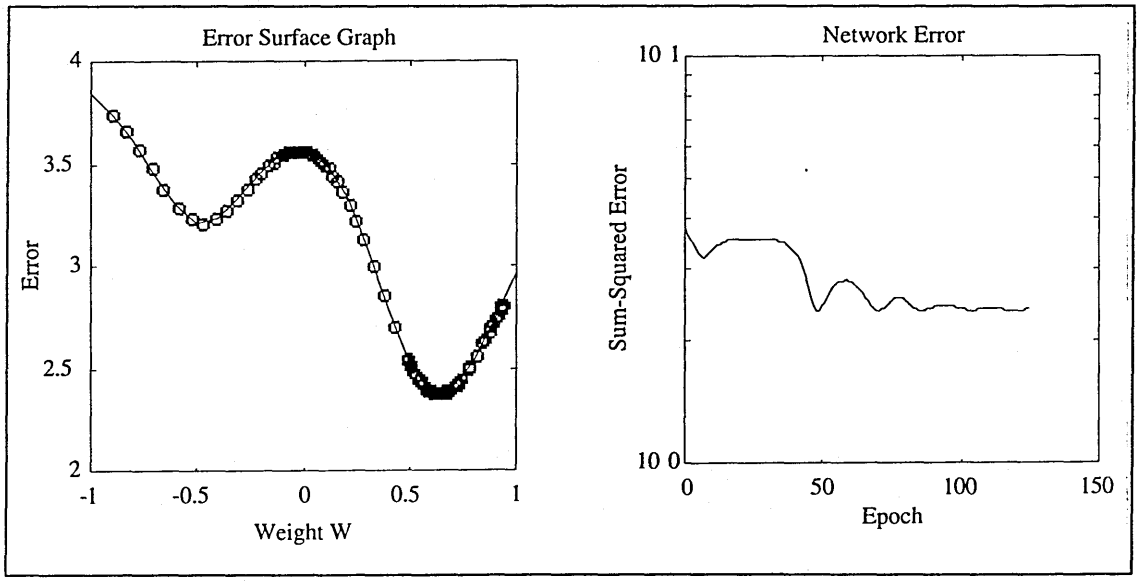

Figure. 2.12 : Avoidance of local error minima using momentum

Denoting the momentum constant of a network with the variable *mc*, the backpropagation learning rules given by (2.22) and (2.23) become:

$$W(i,j)_{new} = W(i,j)_{old} + (1-mc) \times lr \times D(i) \times P(j) \qquad (2.24)$$

$$B(i)_{new} = B(i)_{old} + (1-mc) \times lr \times D(i) \qquad (2.25)$$

A value of between 0 and 1 is chosen for the momentum constant of a network. Thus for a network with a momentum constant of 1, local error information will be ignored and new weight and bias changes will be set equal to the previous weight and bias changes. If a zero momentum constant is used, the learning rules become pure gradient descent algorithms, and previous changes to the network weights and biases are ignored.

2.3.5 Other types of neural networks

So far in this chapter, the class of neural networks that employ some kind of error feedback to update their weights and biases have been considered. The reason for examining these networks in some detail is that it is these types of neural network that have been most commonly applied to control problems. Several other classes of neural networks exist and have found widespread use. Although these types of neural network have been mostly applied to other fields such as pattern recognition, some control applications of these networks have also been published. For this reason, these networks are briefly introduced here.

An important class of neural network that has not been described so far use what are known as associative learning rules. Networks of this type are used to associate particular input vectors with particular output vectors. The common feature of associative learning these rules is that no measure of network error is incorporated into the rule. As networks using associative learning rules do not receive any error information while learning, they are often referred to as "unsupervised learning networks." The most well known neural networks that employ associative learning rules are Kohonen networks [23,24].

A second major class of neural networks that have not so far been discussed are known as Hopfield networks [25-28]. Networks of this type are fully connected (recurrent) and are therefore able to store information. Often, the neurons that are used are the Perceptrons discussed earlier, but the recurrency and complex learning algorithms make these networks quite different in function and ability to basic Perceptrons.

2.3.6 Summary

In this chapter, a basic introduction to the field of neural networks was presented. Emphasis was placed on feedforward types of networks where inputs are fed forward layer by layer to the network output, and then weights and biases updated by some form of feedback of error. These types of networks have been predominantly used in neural network control applications. In the following chapter, some basic neural network control methods are now discussed.

3 Neural Networks for Control

In the previous section, a concise overview of the field of neural networks was presented. In this section, some general neural network control methods are discussed.

3.1 Introduction

The application of neural networks to control problems is an area in which considerable research has been undertaken in recent years [29-34]. The reason why research in this field has become so popular is that neural network control techniques can have a number of advantages over traditional control techniques. These advantages can be summarised as follows:

(i) In many cases, neural network controllers can be trained without the need to obtain a mathematical model of the system to be controlled. This is a particular advantage when the dynamics of a system are complex, and cannot be easily modelled.

(ii) Certain types of neural network can be used as "universal function approximators", that is, they can be trained to provide a functional mapping for any non-linear function [2-5]. The neural network can model the function arbitrarily closely, given a sufficient number of neurons and a long enough training time. This means that neural network control techniques can be particularly useful for controlling non-linear plants, where conventional control methods may be of limited use.

(iii) Training a neural network can be a lengthy process. However, once the network has been trained, it can operate very quickly since the calculations that it carries out are essentially very simple. If a neural network is implemented in hardware, it will operate even faster as the calculations will be carried out in parallel. The speed of operation of neural networks can give them considerable advantages in real-time control applications, where more conventional control algorithms may not be able to be carried out quickly enough.

(iv) The information stored in a neural network is stored as weights between the connections of the individual neurons. The distribution of information in this way results in robust behaviour in the presence of individual component failure.

(v) Neural networks are not programmed with control algorithms, but learn from being trained on a general training data set. This type of learning (which has been called generalised learning) results in neural networks being able to generalise when presented with inputs that are outside their normal mode of operation. This again means that neural network controllers can be more robust than their conventional counterparts.

Although the above summary lists some significant advantages that neural network controllers can have over controllers based on more traditional techniques, one should be aware that there are also a number of potential drawbacks in using neural network technology for control purposes. Some of the problem areas are:

(i) At this stage in neural network technology, there is no set way of producing a controller for a particular control problem. There are a large number of neural network architectures and learning algorithms available, and no real way of knowing in advance which arrangement is likely to work best for the particular problem being considered.

(ii) In the absence of any analytical methods for rigorously deriving an optimum neural network controller for a particular situation, how much faith should one have in the neural network controller that is obtained? This issue is particularly pertinent when a controller is required for a safety critical system, or as is the case in the work in this thesis, for an expensive spacecraft mission.

These issues are developed further in the conclusions section of this thesis, but for now, one should just bear in mind that neural network controllers can have serious disadvantages over their more conventional counterparts.

3.2 Inverse Models

The majority of neural network control techniques share a common goal : to obtain some form of inverse model of the plant to be controlled by training a neural network to model the inverse dynamics of the plant. Once an inverse of the plant has been obtained, it can be placed in the feedback loop of the plant and used for closed loop control.

Figure 3.1 illustrates the principle behind using an inverse model of a plant for control purposes.

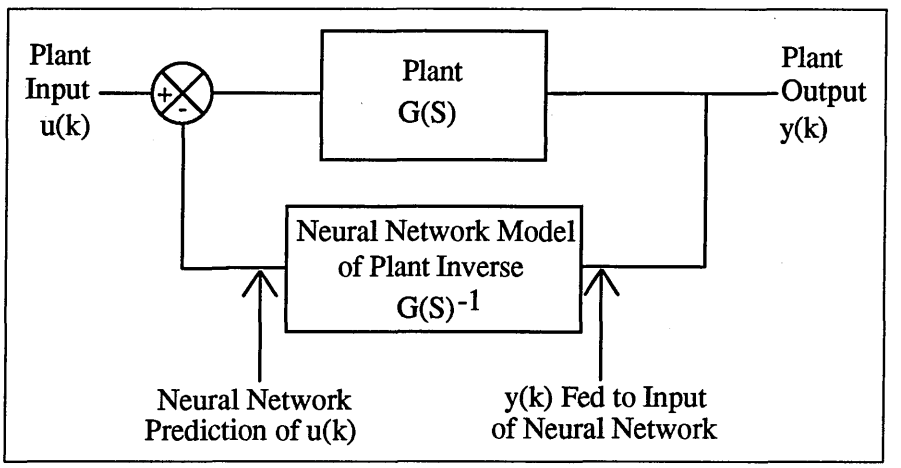

Fig. 3.1 : Using a plant inverse model for closed loop control

In figure 3.1, the transfer function of the system, *F(S)*, is given by

$$F(S) = \frac{G(S)}{1 + G(S)G(S)^{-1}} \tag{3.1}$$

so that

$$F(S) = \frac{G(S)}{2} \tag{3.2}$$

Thus if we have a disturbance input to the plant, we might expect that by using an inverse plant model in the feedback loop, we might reduce the effect of the disturbance at the output of the plant by approximately one half. Clearly, further improvements in performance will be obtained if a gain term is also included in the feedback path, and the gain increased as much as possible, subject to the plant remaining stable.

As most neural networks are based on obtaining an inverse plant model, the main differences between the various neural network control techniques concern the choice of the following parameters :

(i) Neural network architecture (number of neurons, type of transfer functions used)

(ii) Form of the learning algorithm used for updating network weights

(iii) Values of parameters used in the learning algorithm

(iv) On-line / off-line adjustment of network weights

(v) Training procedure for obtaining an inverse plant model

(vi) Variables used as inputs for the inverse model neural network

In the following sections, two alternative methods for obtaining an inverse plant model are considered.

3.3 Direct Model Inversion

Direct Model Inversion is the name given to the most straightforward of the methods that have been suggested for obtaining an inverse plant model. In this method the following procedure is adopted:

(i) Generate a training data set consisting of plant input-output vector pairs, as illustrated in figure 3.2

(ii) Present the plant output vectors to the inputs of the network and train it to faithfully produce the corresponding plant input vectors at its outputs. This procedure is illustrated in figure 3.3

The input-output vector pairs used to train the network can be obtained by constructing a mathematical model of the plant to be controlled, and performing a computer

simulation to produce theoretical input-output pairs. Alternatively, a training data set can be obtained from the actual plant itself, if this available. Note that in the former case, the quality of the neural network controller produced will be strongly dependent on how closely the mathematical model that is constructed actually represents the behaviour of the real plant.

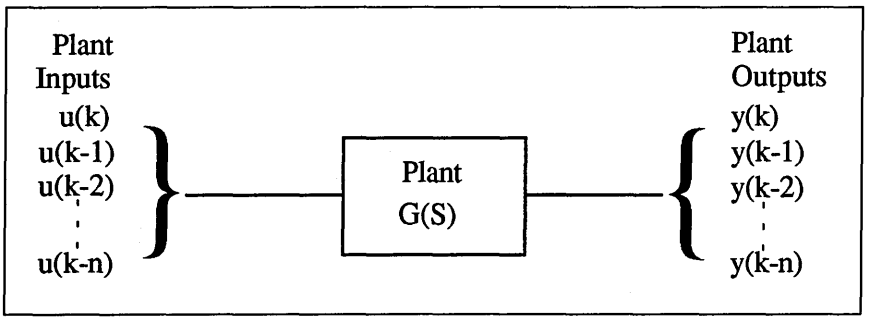

Fig. 3.2 : Generation of a training vector set

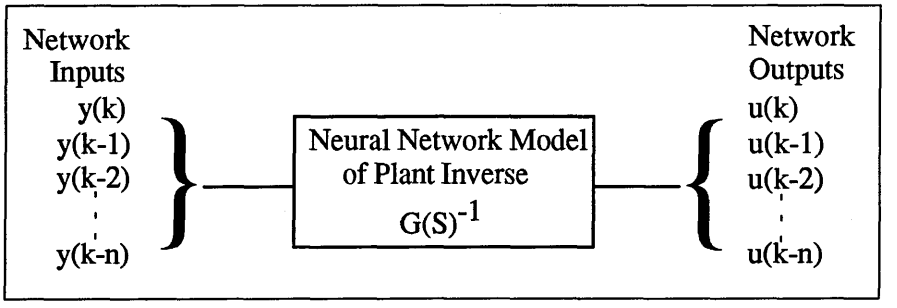

Fig 3.3 : Presentation of training vectors using the Direct Model Inversion method

An important consideration to make when constructing an inverse model using a method such as direct model inversion is what variables should be used as inputs to the inverse model network. If the network is supplied with only the current plant output y(k), it will be unable to determine the current plant input u(k) as it will not have been provided with any information about the current state of the plant. To get around this problem, other information must be provided at the inputs of the inverse model network, such as time derivatives of plant outputs, or possibly just a number of previous plant outputs. In the work carried out in this thesis, the current plant output, y(k), together with the last seven plant outputs, y(k-1) to y(k-7) were used as network inputs as time derivatives of these plant outputs were not considered to be readily available. This is discussed in more detail in chapter 5.

Note that one of the possible disadvantages of the Direct Model Inversion method is that it is essentially an off-line technique. A training data set is produced and a neural

network controller trained before actually being used to control the plant. The network controller is therefore unable to update the weights of its connections during the course of its operation. The technique does not, therefore, lend itself to self-adaptation in the presence of unexpected changes in the dynamics of the plant.

3.4 Indirect Model Inversion

The Indirect Model Inversion method is similar to the Direct Model Inversion method in that the objective once again is to obtain a neural network that broadly models the inverse dynamics of the system to be controlled. The method differs, however, in that it requires the use of two networks. The first is a forward model of the plant to be controlled and is trained off-line using a training data set of input-output pairs. The second network is the controller network, and is trained on-line in accordance with both the performance of the network as a controller and the parameters chosen for the network's learning algorithm.

The Indirect Model Inversion method involves adopting the following procedure :

(i) A set of plant input-output vector pairs is obtained from the actual plant or from a simulation.

(ii) A neural network is trained off-line to model the forward dynamics of the plant to be controlled, using the input-output data set obtained in step (i).

(iii) The initial weights and biases of a second network (which will be the controller network) are randomised. The network is then connected according to the configuration shown in figure 3.4.

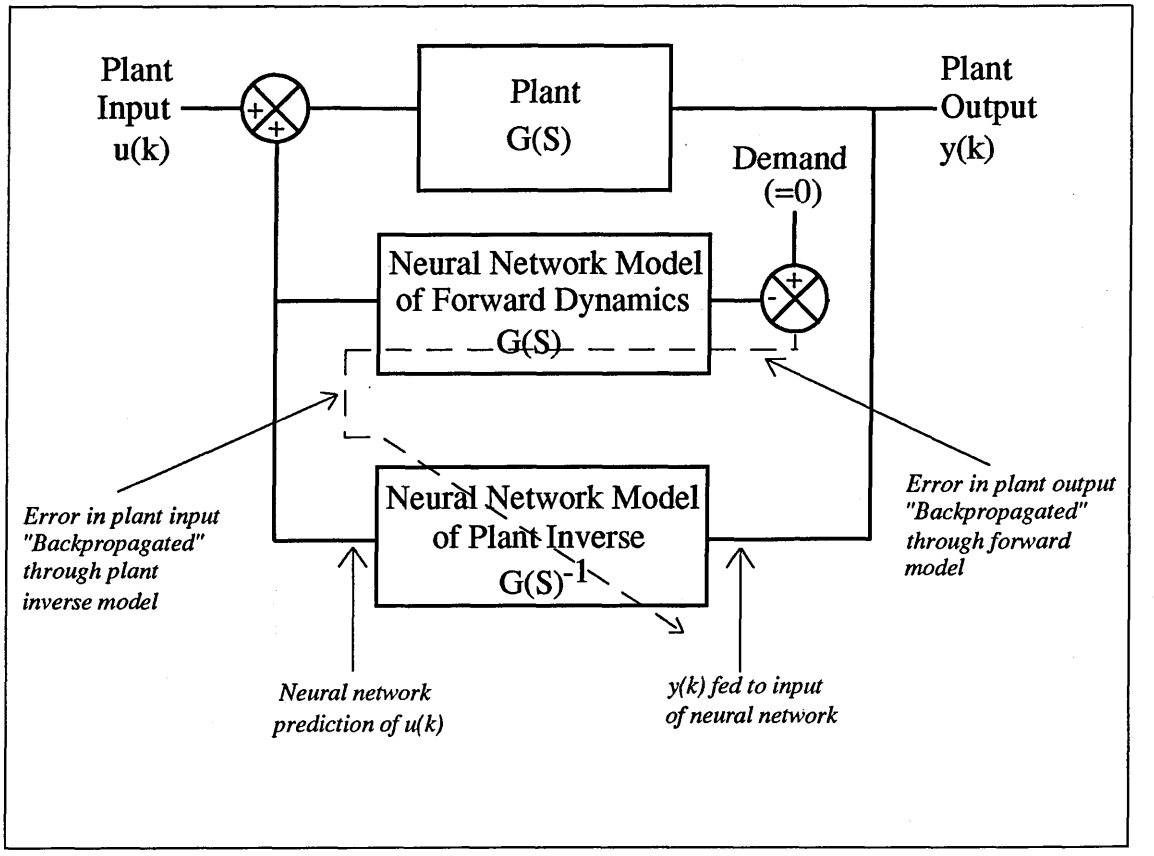

Figure 3.4 : Schematic representation of the Indirect Model Inversion method

(iv) The control system is turned on, and errors calculated in the plant output from the output of the forward model neural network.

(v) The errors in the plant output obtained in step (iv) are backpropagated (passed back) through the forward model network, but without adapting the weights and biases of this network. This serves to map the plant error from "plant output space" to "plant input space".

(vi) The error in "plant input space" that was obtained in step (v) is then backpropagated through the inverse model network, and the weights and biases updated to minimise this error.

Note that in the Indirect Model Inversion method, the role of the forward model neural network is to obtain an appropriate error signal for use by the learning algorithm of the controller (inverse model) neural network. Thus, the forward model serves to simply map the errors in the plant output to corresponding errors in the plant input.

As the adjustment of the weights and biases of the controller network is dependent on the performance of the controller, the Indirect Model Inversion technique is really inherently an on-line technique. Thus a neural network controller system can be configured so that it either learns for a specified time, or so that it learns throughout its lifetime with network weights being continually updated. In the latter configuration, the resulting controller system can be particularly robust, as future unexpected changes in the dynamics of the plant can be accounted for by the controller.

Although it has just been pointed out that the indirect model inversion method is inherently an on-line technique, an off-line equivalent is also available. The procedure is as follows :

(i) A set of plant input-output vector pairs are obtained from the actual plant or from a simulation.

(ii) A neural network is trained off-line to model the forward dynamics of the plant to be controlled, using the input-output data set obtained in step (i).

(iii) The initial weights and biases of a second network that is to be the controller (inverse model) network are randomised.

(iv) The plant output vectors obtained in step (i) are passed to the controller network and the controller outputs stored.

(v) The controller outputs obtained in step (iv) are then added to the plant inputs used in step (i), and the vectors that result are presented to the plant.

(vi) The output of the plant (the plant error) obtained in step (v) is backpropagated through the forward model neural network to obtain an error in "plant input space".

(vii) The weights of the controller network are updated in such a way that the error in "plant input space" obtained in step (vi) is minimised.

This procedure is effectively a batched version of the on-line Indirect Model Inversion method discussed at the beginning of the section.

The main advantage of the procedure is that the controller error is minimised for a batch of vectors rather than for a single vector at a particular instant in time, which is normally the case using the on-line method. In this way, the controller can adapt in order to minimise a global error that represents a range of plant conditions. This is quite different to the on-line method, where the controller adapts to a local error that applies to a particular plant state at a particular instant in time.

The main disadvantage of the off-line method compared to the on-line method is that the controller obtained cannot be adapted later on-line, and therefore is less robust in the face of unexpected changes in the dynamics of the plant.

Although the off-line version of the Indirect Model Inversion method is very much simpler to implement than its on-line counterpart, it was not investigated during the course of this work. This is because none of the papers that were read as part of the initial literature survey made reference to the off-line method, and its potential advantages only became apparent at a late stage of this work, in the light of disappointing results obtained using the on-line method. However, it is unclear whether improved results would have been obtained if this procedure was used.

3.5 Suitable Neural Network Architectures

The goal in neural network control applications is to train a network to map plant output errors to suitable control inputs. Often, the function that describes this mapping is complex and non-linear, and is not easily described using conventional control theory.

In view of the requirement that neural networks in control applications learn a particular function, it is the neural networks that are known to have good function approximation capabilities that have been used most frequently for control purposes. As already stated, backpropagation networks of sigmoid neurons, and Radial Basis Function (RBF) networks are well known to provide excellent function approximation abilities [2-5,10].

In the work carried out in this thesis, backpropagation networks with tansigmoidal hidden neurons and linear output layers were used, as networks such as these were readily supported by the Matlab Neural Network Toolbox that was to be used for this work.

4 Spacecraft Plant Model

In order to investigate how neural network control techniques can be applied to a spacecraft control problem, a realistic computer model of a spacecraft must first be constructed for simulation purposes.

The software that was to be used for this study comprised the Matlab mathematical software combined with the Simulink control system simulation package and the Matlab Neural Network Toolbox. Given that Simulink was to be used, it was necessary only to obtain suitable transfer functions for the components of the spacecraft model, as these can then be simply linked using the Simulink graphical user interface to construct a complete computer model of the spacecraft ready for simulation.

As a basic starting point, it was decided to use a simplified version of the SOHO spacecraft dynamics, as represented in the work carried out by [6]. In this scheme, the dynamics of the SOHO spacecraft are considered only about a single axis, and can be represented by a series of single-input single-output transfer functions.

The simplified model of the SOHO dynamics given in [6] was then extended to include a transfer function that was to represent the effect of spacecraft fuel sloshing.

In all, the following components were built into the spacecraft model :

(i) Disturbance torques due to a mass imbalance in one of the reaction wheels of the spacecraft.

(ii) Rigid body spacecraft inertia

(iii) Solar array flexure

(iv) Payload vibration

(v) Spacecraft fuel slosh

In the following subsections, each of these components is considered in turn.

4.1 Disturbance Torques

The source of the disturbance torques acting on the spacecraft in this work is a mass imbalance in one of the reaction wheels of the spacecraft. This source was chosen in accordance with work carried out by [6] as the results produced in this paper were initially reproduced as a starting point for the work in this thesis.

Although a reaction wheel mass imbalance was chosen as the source of the disturbance torques acting on the spacecraft throughout this work, the control problem considered is by no means restricted to this choice. In fact, there is no shortage of possible candidates that could have equally well have been chosen to provide a disturbance, e.g., thruster misalignment, thermal vibrations, movement of internal mechanisms etc. It really would not have mattered which of these was chosen, just as long as a realistic disturbance torque was fed to the input of the spacecraft model.

The disturbance torque T_d created by a mass imbalance in a spacecraft reaction wheel rotating at ω rad s^{-1} is given by

$$T_d = k\omega^2 \sin(\omega t + \phi) \tag{4.1}$$

where k is a constant term that includes the inertia of the wheel (4.5×10^{-6} kgm^2 for the SOHO spacecraft) and the mass imbalance of the wheel.

In [6] a feasible peak disturbance torque created by a mass imbalance in a SOHO reaction wheel rotating at 60 $rads^{-1}$ is given as 0.006 Nm. This then gives a value of k of 1.67×10^{-6} kgm^2. The equation used in this work to model disturbance torques in the Simulink environment was therefore :

$$T_d = 1.67 \times 10^{-6} \omega^2 \sin(\omega t) \tag{4.2}$$

where the phase, ϕ, is ignored as we are only considering disturbances produced by one of the SOHO reaction wheels.

4.2 Spacecraft Inertia

The main component of the model of the dynamics of the spacecraft under consideration is the rigid body inertia of the spacecraft about a single axis. This component is also very simple to model.

When the disturbance torque T_d acts on a spacecraft with rigid body of inertia I, the angular acceleration $\ddot{\theta}_s$ of the spacecraft is given by

$$\ddot{\theta}_s = T_d \times \frac{1}{I} \tag{4.3}$$

Thus if no other dynamical effects are present, the angular acceleration of the spacecraft model about the axis under consideration can be obtained by feeding the disturbance torque to a $1/I$ transfer function, as illustrated in figure 4.1.

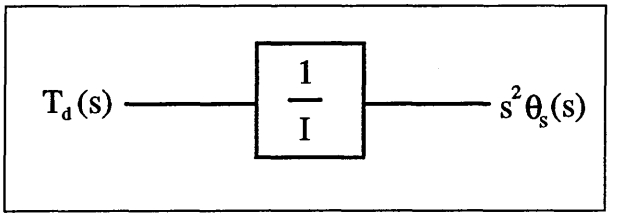

Fig. 4.1 : Block diagram of a 1-component spacecraft model

The rigid body inertia of the SOHO spacecraft about the axis under consideration in this work is 2600 kgm^2. The spacecraft inertia was therefore represented in the Simulink environment using a 1/2600 gain term.

4.3 Solar Array Flexure

In view of the need to minimise spacecraft mass and to maximise the available on-board electrical power, solar arrays tend to be low mass structures with large panel areas. Structures with these characteristics tend to have considerable flexibility. It is therefore important that the effect of solar array flexure is properly taken into consideration if any spacecraft control system analysis work is to be carried out.

There are various types of model that can be used to represent the effects of solar array flexure. The two main types of model are

(i) The cantilever model
(ii) The free-free model

In the cantilever model, the solar arrays are treated as flexible beams which are restrained from motion at their roots (i.e. the main body of the spacecraft is considered to be fixed in this model).

In the free-free model, the roots of the solar arrays are not assumed to be restrained from motion, and the natural vibration modes of the whole spacecraft structure are therefore considered (i.e. the main body of the spacecraft is no longer considered to be fixed).

Clearly, the dynamics of solar array flexure are likely to be more accurately represented if the free-free model is used instead of the cantilever model. However, the free-free model is more difficult to implement. An even greater level of accuracy can be achieved if a finite element package such as NASTRAN is used, but the simulations carried out during the course of this work did not require the higher level of accuracy that would result from the adoption of NASTRAN or the free-free model. For these reasons, the cantilever model was used to model solar array flexure in the simulations in this work.

Using the cantilever model, the overall spacecraft dynamics in the presence of solar array flexure can be described by the following two equations :

$$T_d = I\ddot{\theta}_s - \sum_i \delta_{ai}\ddot{\eta}_{ai} \tag{4.4}$$

and

$$\delta_{ai}\ddot{\theta}_s = \ddot{\eta}_{ai} + 2\zeta_{ai}\omega_{ai}\dot{\eta}_{ai} + \omega_{ai}^2\eta_{ai} \tag{4.5}$$

In the above equations, T_d is the disturbance torque acting on the spacecraft, θ_s is the attitude of the spacecraft as shown in fig 4.1, δ_{ai} is the coupling coefficient of the i^{th} flexure mode of the solar arrays, and ζ_{ai} and ω_{ai} are the damping ratio and natural frequency of the i^{th} flexure mode (i.e. each flexure mode of the arrays is treated as a separate second order system). η_{ai} is the modal co-ordinate of the i^{th} array flexure

mode. It is a parameter that allows us to transform the equations for the dynamics of solar array flexure from the solar array reference frame (in which the spacecraft is considered to be fixed) to a frame of reference in which the motions of the spacecraft can be considered.

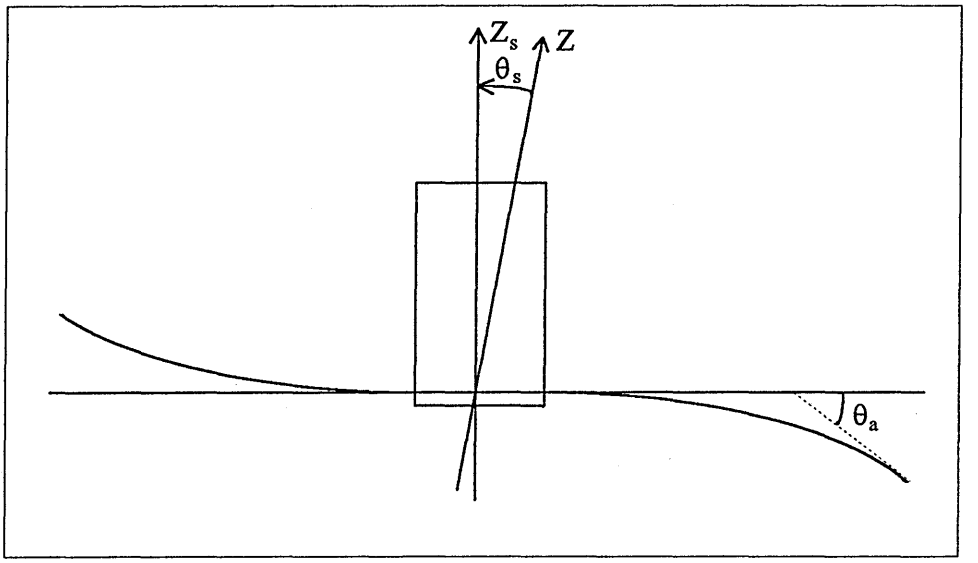

Fig. 4.2 : Cantilever model of solar array flexure showing first flexure mode

The effects of the various flexure modes can be summed in the fashion shown in equation 4.4 as long as only small angular deformations are considered, that is, as long as the dynamics of solar array flexure can be considered to be linear. However, as high accuracy was not a prerequisite for the simulations carried out in this work, only the first flexure mode of the solar arrays was taken into consideration.

By combining equations (4.4) and (4.5) and rearranging, we can obtain a transfer function for the overall system. Taking Laplace Transforms of (4.4) and (4.5) and considering only the first flexure mode gives

$$T_d(s) = s^2 I\theta_s(s) - s^2\delta_a\eta_a(s) \tag{4.6}$$

and

$$s^2\eta_a(s) + 2\zeta_a\omega_a s\eta_a(s) + \omega_a^2\eta_a(s) = s^2\delta_a\theta_s(s) \tag{4.7}$$

Substituting (4.7) into (4.6) we get :

$$T_d(s) = s^2 I\theta_s(s) - \frac{s^4\delta_a^2\theta_s(s)}{s^2 + 2\zeta_a\omega_a s + \omega_a^2} \tag{4.8}$$

Rearranging, we obtain the transfer function for the overall system relating the angular acceleration of the main body of the spacecraft to the input disturbance torque:

$$\frac{s^2\theta_s(s)}{T_d(s)} = \frac{1}{I - \dfrac{s^2\delta_a^2}{s^2 + 2\zeta_a\omega_a s + \omega_a^2}} \tag{4.9}$$

We now need to consider how equation 4.9, which represents the rigid body dynamics and the solar array dynamics of the spacecraft, can be represented in block diagram form, suitable for use in the Simulink control system simulation environment. As a first starting point, we could assume that the block diagram will take the same form as that shown in figure 4.1, but with an additional transfer function in a feedback loop which would represent the effects of solar array flexure. Such a block diagram is shown in figure 4.3, where the effects of rigid body inertia and solar array flexure are represented by the transfer functions A and B.

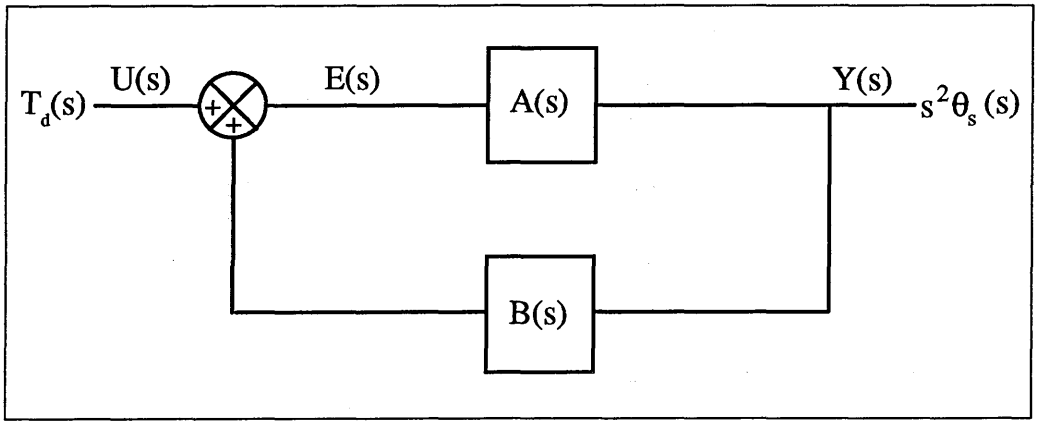

Fig. 4.3 : Generalised block diagram of a 2-component spacecraft model

The input to transfer function A in figure 4.3 is given by

$$E(s) = U(s) + B(s)Y(s) \tag{4.10}$$

and the output of transfer function A is given by

$$Y(s) = A(s)E(s) \tag{4.11}$$

Combining these equations gives

$$\frac{Y(s)}{A(s)} = U(s) + B(s)Y(s) \tag{4.12}$$

which gives us

$$Y(s)\left(\frac{1}{A(s)} - B(s)\right) = U(s) \tag{4.13}$$

and so we can see that the transfer function for the block diagram given in figure 4.3 can be written as

$$\frac{Y(s)}{U(s)} = \frac{1}{\frac{1}{A(s)} - B(s)} \tag{4.14}$$

Comparing equations (4.9) and (4.14), it can be seen that the block diagram illustrated in figure 4.2 will represent the dynamics of the spacecraft if

$$A(s) = \frac{1}{I} \tag{4.15}$$

(as before) and

$$B(s) = \frac{s^2\delta_a^2}{s^2 + 2\zeta_a\omega_a s + \omega_a^2} \tag{4.16}$$

It can therefore be seen that the simple spacecraft model developed in the previous section (in which only rigid body inertia was represented) can be extended to incorporate the effect of solar array flexure by including a suitable transfer function in a positive feedback loop. The new block diagram for the system is shown in figure 4.4

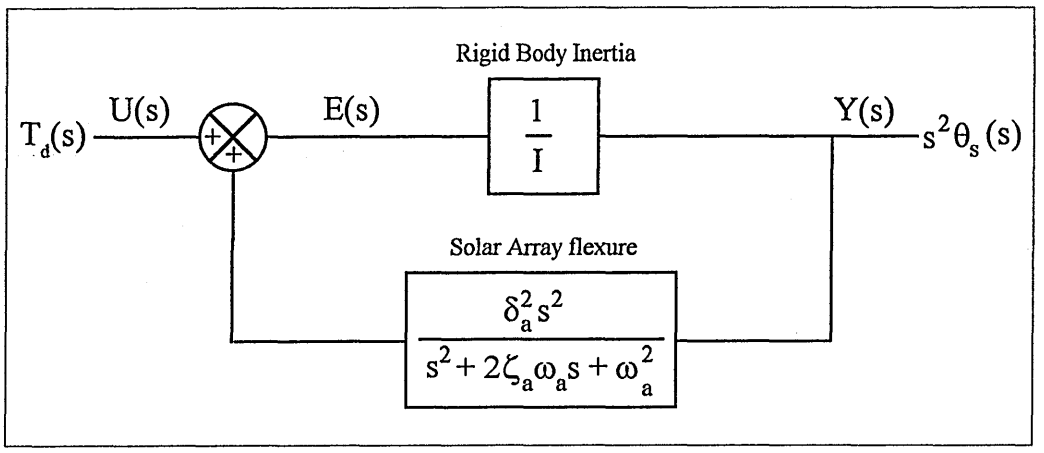

Fig. 4.4 : Block diagram of a 2-component spacecraft model

For the solar arrays on the SOHO spacecraft, the natural frequency and damping ratio for the first flexure mode are given by

$$\zeta_a = 0.005 \tag{4.17}$$

$$\omega_a = 3.7 \text{ rad s}^{-1} \tag{4.18}$$

and the coupling coefficient for the first mode is given by

$$\delta_a = 20 \text{ kg}^{\frac{1}{2}}\text{m} \tag{4.19}$$

Thus the transfer function used in the positive feedback loop representing solar array flexure was :

$$\frac{400s^2}{s^2+0.037s+13.69} \tag{4.20}$$

4.4 Payload Vibration

The spacecraft model obtained in the previous section can be simply extended to take into account the dynamics of payload vibration. The approach that was used to incorporate the effect of solar array flexure to the initial model (in which only rigid body inertia was considered) can be adopted once again.

As with the solar arrays, the payload can be treated as an independent second order system, and coupled to the dynamics of the main body of the spacecraft using a coupling coefficient with the appropriate magnitude. Once again, only the first flexure mode need be considered. Thus, the transfer function representing payload deflection can be written as

$$C(s) = \frac{s^2\delta_p^2}{s^2 + 2\zeta_p\omega_p s + \omega_p^2} \tag{4.21}$$

which is of the same form as (4.16). This transfer function is incorporated into the rest of the model by inserting it into an additional positive feedback loop, in parallel with the transfer function representing solar array flexure.

In the previous section, we were not concerned with monitoring the angular position of the solar arrays, as we needed only to consider how the flexure interacted with the rigid body spacecraft dynamics. However, this is not so in the case of the spacecraft payload. This is because the objective of this work is to use a neural network controller to minimise payload deflection in the presence of a disturbance torque. It is therefore necessary for the controller to have knowledge of the angular position of the payload.

As the spacecraft model considered in these sections is broadly based on the SOHO spacecraft, we could assume that the payload under consideration is an optical payload. We could then also assume that a measure of the attitude of the payload will be available from the measurements made by the payload itself, and that no additional attitude sensors are required.

We now need to consider how the transfer function given in equation (4.21) can be used to obtain angular positions of the payload tip. So far, we have a spacecraft model that incorporates rigid body inertia, solar array flexure and payload flexure. A generalised block diagram of this model is given in figure 4.5, where A, B and C are given by equations (4.15), (4.16) and (4.21) respectively.

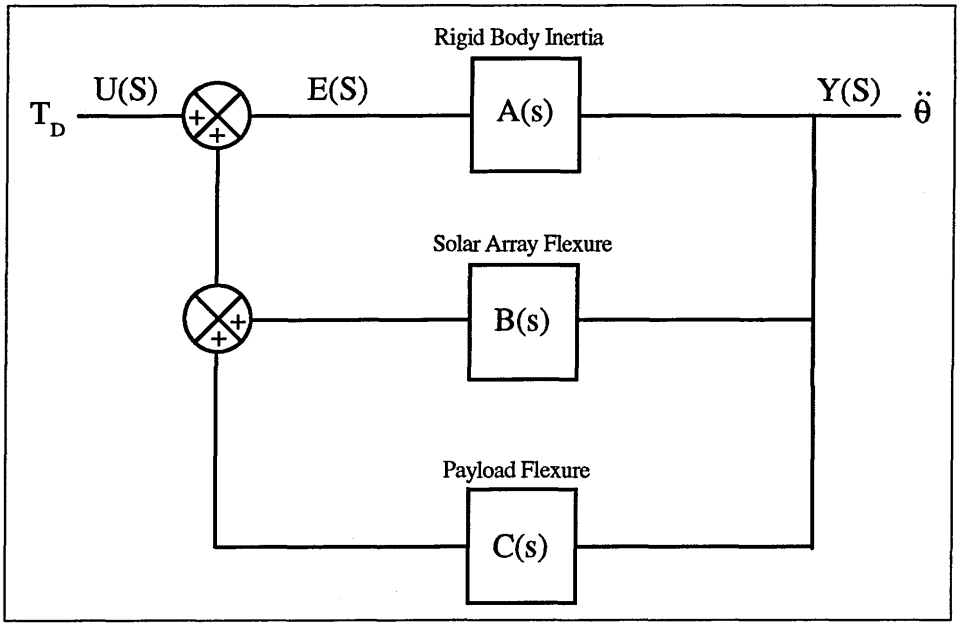

Fig. 4.5 : Generalised block diagram of a 3-component spacecraft model

The first step is to replace transfer function C(s) with two transfer functions, $C_1(s)$and $C_2(s)$, and assume that some information concerning payload deflection is available between the two. This configuration is illustrated in figure 4.6.

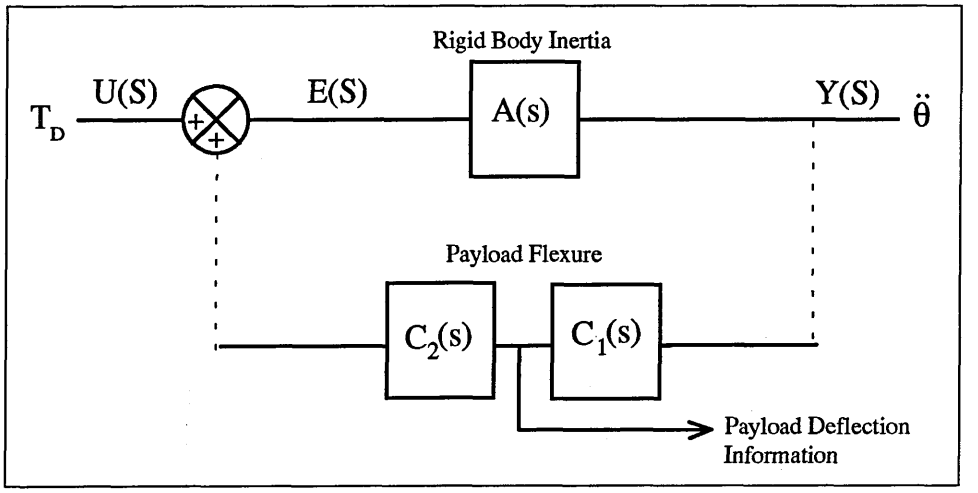

Fig. 4.6 : Extraction of payload deflection information from the spacecraft model

As the cantilever model has been chosen to model the flexure of various components aboard the spacecraft, the angular deflection of the payload is not directly available from the block diagram as shown in figure 4.4. However, the modal co-ordinate η_p is available, and this can then be used to calculate the angular payload deflection.

If we choose the payload deflection information in figure 4.6 to be the modal co-ordinate η_p, then the values of the transfer functions C_1 and C_2 are described by the equations

$$C_1(s)C_2(s) = \frac{\delta_p^2 s^2}{s^2 + 2\zeta_p \omega_p s + \omega_p^2} \tag{4.22}$$

and

$$C_1 = \frac{\eta_p}{s^2 \theta_s} \tag{4.23}$$

From equation (4.7) we see that for the first flexure mode of the payload,

$$s^2\eta_p + 2\zeta_p\omega_p\eta_p s + \omega_p^2\eta_p = \delta_p\theta_s \tag{4.24}$$

giving

$$\eta_p = \frac{s^2\delta_p\theta_s}{s^2 + 2\zeta_p\omega_p s + \omega_p^2} \tag{4.25}$$

Substituting (4.25) into (4.23) then gives

$$C_1(s) = \frac{\delta_p}{s^2 + 2\zeta_p\omega_p s + \omega_p^2} \tag{4.26}$$

Finally, substituting (4.26) into (4.22) gives us

$$C_2(s) = \delta_p s^2 \tag{4.27}$$

Thus, the overall block diagram of the system is now as shown in figure (4.7)

Next we must consider how we can obtain the payload deflection angle θ_p from the modal co-ordinate. The procedure is to calculate the linear deflection first, and then to calculate the angular deflection from the linear deflection.

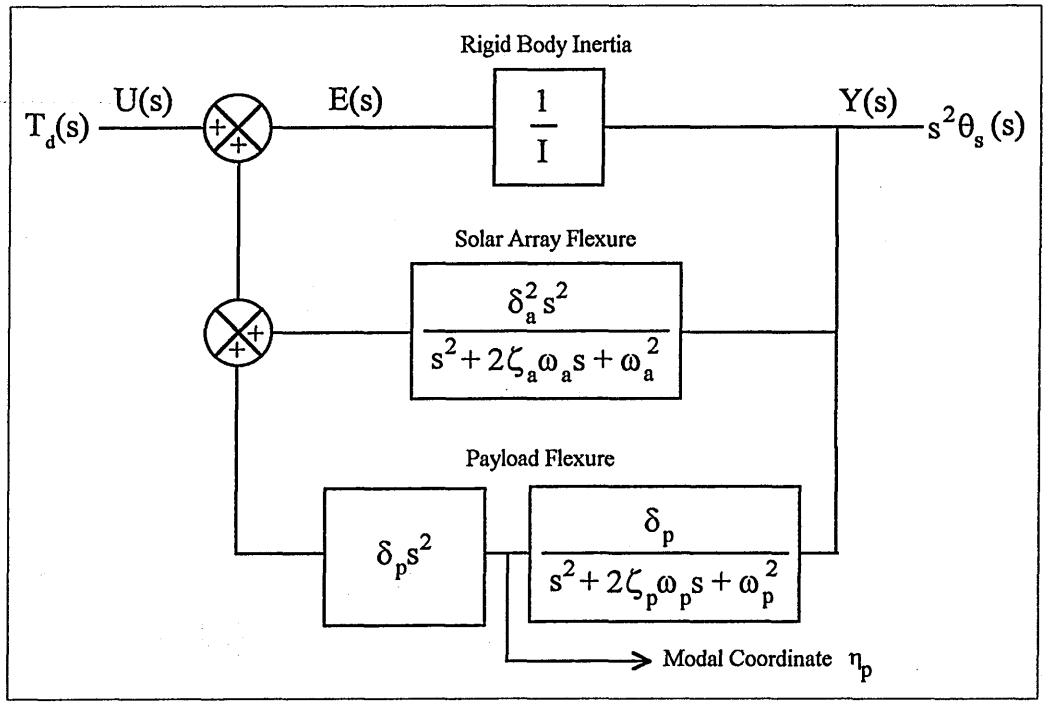

Fig. 4.7 : Block diagram of a 3-component spacecraft model

The relationship between the linear deflection of the payload tip y_p and the modal co-ordinate η_p is given by

$$y_p = \left(\frac{f(x)}{\left(\int \rho(x) f^2(x) dx \right)^{\frac{1}{2}}} \right) \eta_p \tag{4.28}$$

where $f(x)$ is the shape of the mode and $\rho(x)$ is the mass per unit length of the payload. The relationship between the linear deflection of the payload tip y_p and the angular deflection θ_p is given by

$$y_p = k\theta_p \tag{4.29}$$

where k will depend on the shape of the mode and the length of the payload.

As the work carried out in this thesis does not require a high level of accuracy for the spacecraft model, we need not concern ourselves with the mass and length of a particular payload or with the precise shape of the first flexure mode. Instead, we can make certain simplifying assumptions. It turns out that for a range of mode shapes and payloads, the bracketed expression in (4.28) and the constant term k in (4.29) are of order unity. For the sake of simplicity, the magnitudes of the linear deflection y_p and the modal co-

ordinate η_p could be assumed to be equal. However, to remain consistent with the work published in [6] (the results of which were reproduced as a starting point for the work carried out in this thesis) the following relation was used :

$$\theta_p = \left(4.8 \text{ kg}^{-\frac{1}{2}} \text{ m}^{-1}\right)\eta_p \tag{4.30}$$

For the payload assumed for the spacecraft model, the damping ratio, natural frequency and coupling coefficient for the first flexure mode of the payload are given respectively by :

$$\zeta_p = 0. \tag{4.31}$$

$$\omega_p = 63 \text{ rad s}^{-1} \tag{4.32}$$

$$\delta_p = 10 \text{ kg}^{\frac{1}{2}}\text{m} \tag{4.33}$$

Thus the two transfer functions that are used in a positive feedback loop to represent payload flexure are given by :

$$\frac{10}{s^2 + 12.6s + 3969} \tag{4.34}$$

and

$$10s^2 \tag{4.35}$$

4.5 Fuel Sloshing

In view of the prevalent use of liquid propellant for spacecraft orbit and attitude control, there is a strong need for spacecraft engineers to understand the dynamic effects of contained liquids.

The two most significant areas in which the effects of liquid slosh must be given adequate consideration are :

(i) Nutational instability due to propellant energy dissipation (applicable to spin stabilised spacecraft)

(ii) The interaction of fuel slosh with spacecraft attitude control systems

In the work carried out in this thesis, it is this latter effect that must be taken into consideration.

The hydrodynamical equations governing the motions of liquids are very complex, particularly when motion in a low gravity environment is considered. Thus in the absence of analytical solutions, some form of model must be constructed. There are two main approaches that are generally adopted.

In the first method, a scale model of the spacecraft under consideration is built and tested on the ground, and the results are then extrapolated to in-orbit conditions using dimensional analysis.

The second method relies on assuming that the fuel aboard the spacecraft comprises two components : a fixed fuel mass and a slosh mass. A spherical pendulum attached to a spring-damper system is then used to represent the slosh mass. In this way the effect of fuel sloshing can be represented as a second order system.

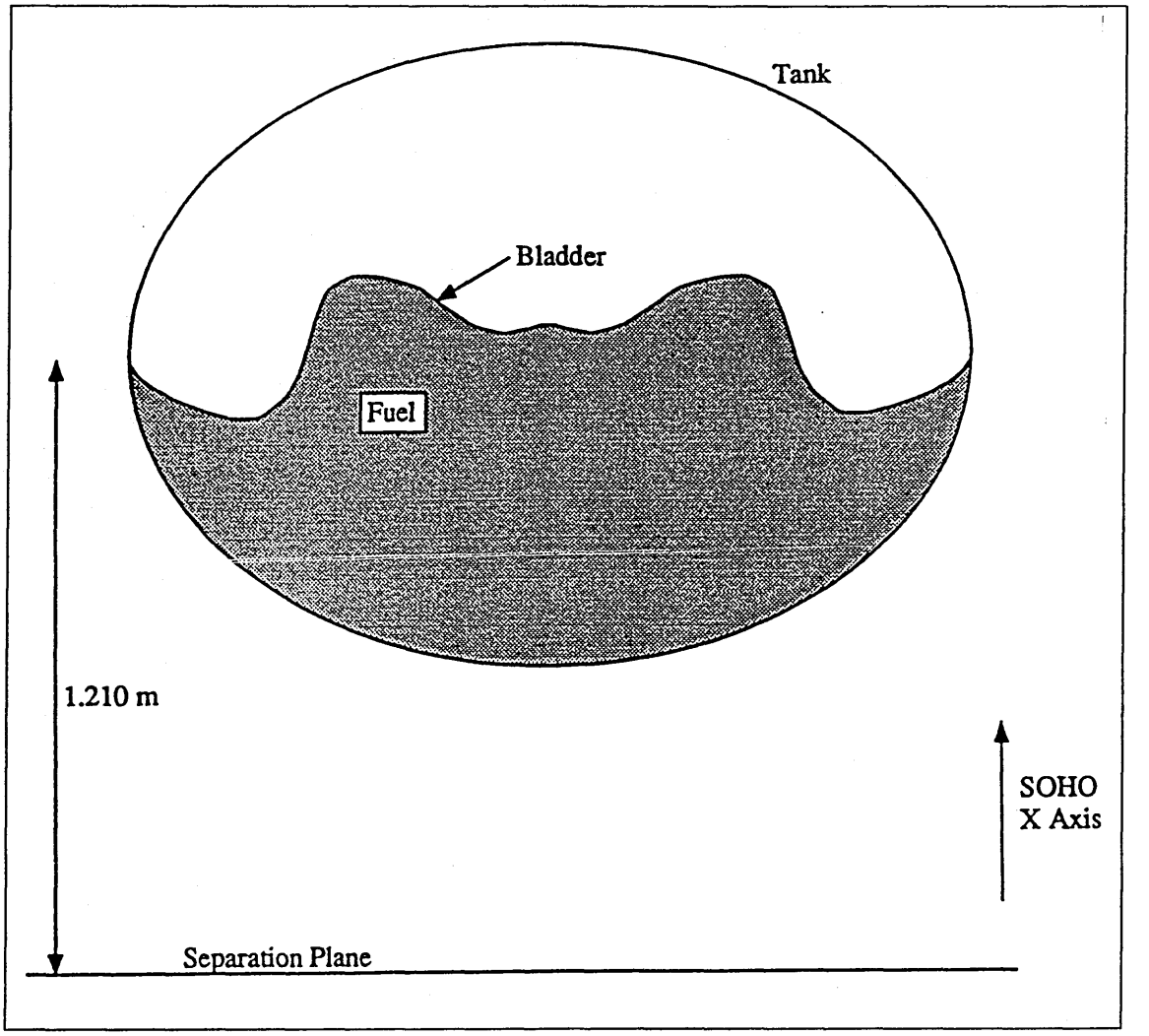

Fig. 4.8 : The PROS fuel tank

The fuel tank aboard the SOHO spacecraft is the PROS fuel tank which was initially designed for the TDRSS program. This tank has a polymeric bladder attached halfway up which restricts the motion of the fuel and hence reduces the effect of fuel sloshing. A diagram of the PROS fuel tank is shown in figure 4.8. Its location within the SOHO spacecraft is shown in figure 4.9.

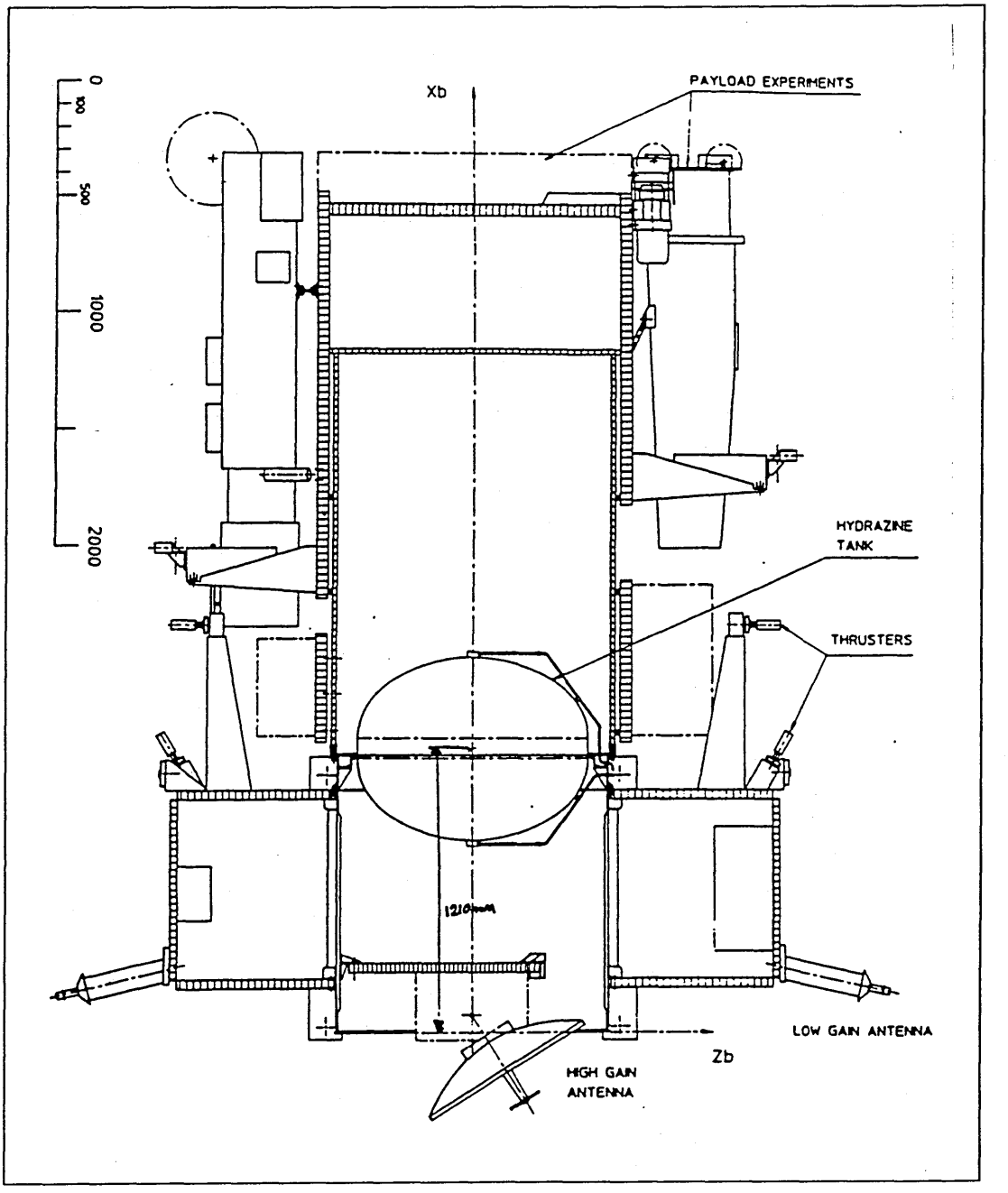

Fig. 4.9 : The SOHO spacecraft, showing the location of the PROS fuel tank

The effect of fuel sloshing aboard the SOHO spacecraft has been extensively modelled by BAe Space Systems Limited, Earth Observation and Science Division, as part of the European Space Agency SOHO project. In this model the effect of fuel sloshing is represented in each axis by two second order penduli. This model, in its exact form, was inappropriate for use in the work carried out in this thesis for the following reasons :

(i) Work carried out by BAe Space Systems Limited on the dynamics of fuel sloshing aboard the SOHO spacecraft indicates that the effect is likely to be small. Thus the effect is unlikely to pose a great problem for either conventional or neural network-based controllers.

(ii) The modelling of the fuel slosh dynamics is fairly complex, and includes considerable cross-coupling between the three axes of the spacecraft (i.e. between the pairs of penduli in each of the spacecraft axes). The BAe model cannot, therefore, be easily incorporated into the above scheme and entered into Simulink environment.

For these two reasons, it was decided that the effect of fuel sloshing on the SOHO spacecraft should be modelled in much the same way as the effects of solar array flexure and payload vibration were modelled in the previous two sections, but with the values of some of the parameters in the BAe model included.

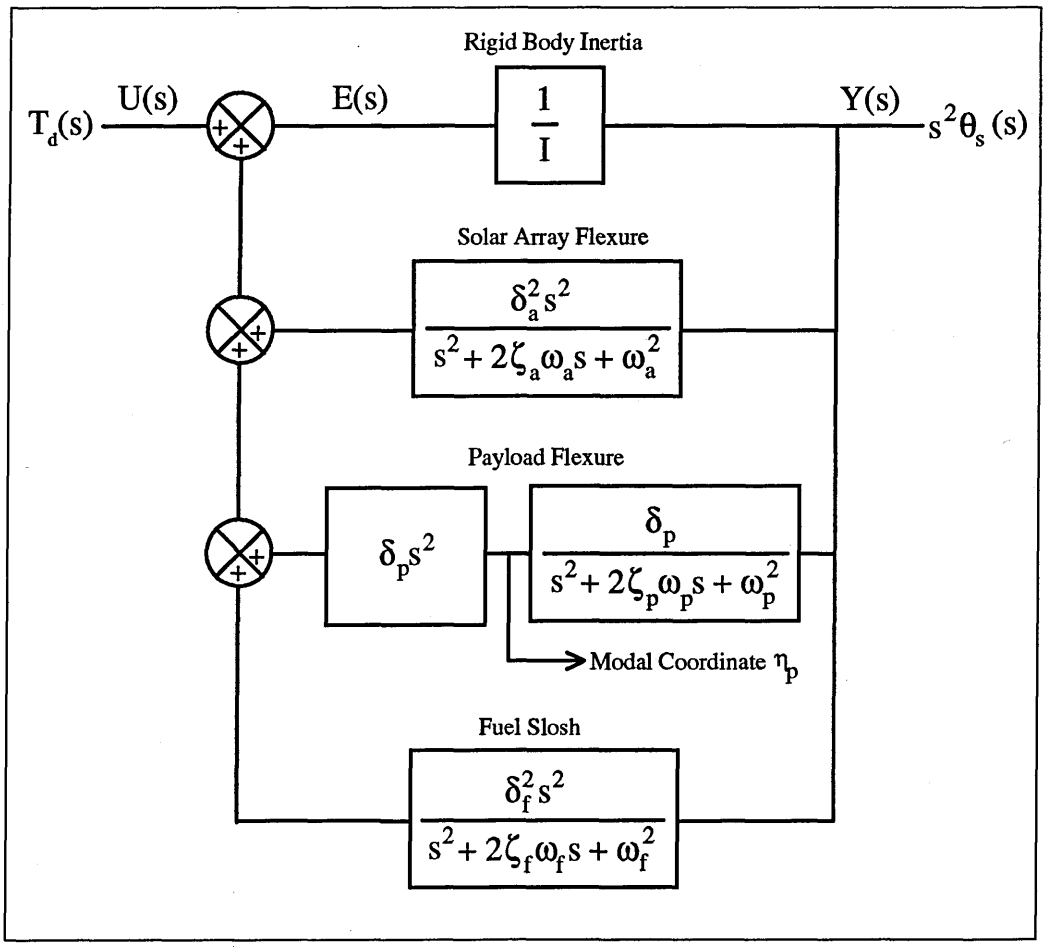

Fig. 4.10 : Block diagram of the complete spacecraft model

Continuing the modelling scheme developed in the previous two sections, the effect of fuel slosh can be represented by a transfer function of the form

$$\frac{\delta_f^2 s^2}{s^2 + 2\zeta_f \omega_f s + \omega_f^2} \tag{4.36}$$

This transfer function can then be inserted into a third positive feedback loop, as illustrated in figure 4.10.

The pendulum model that has been used in the BAe model for +X manoeuvres is shown in figure 4.11. For a spacecraft fill ratio of 52%, the values of the parameters shown in figure 4.11, according to the BAe model, are as shown in table (4.1).

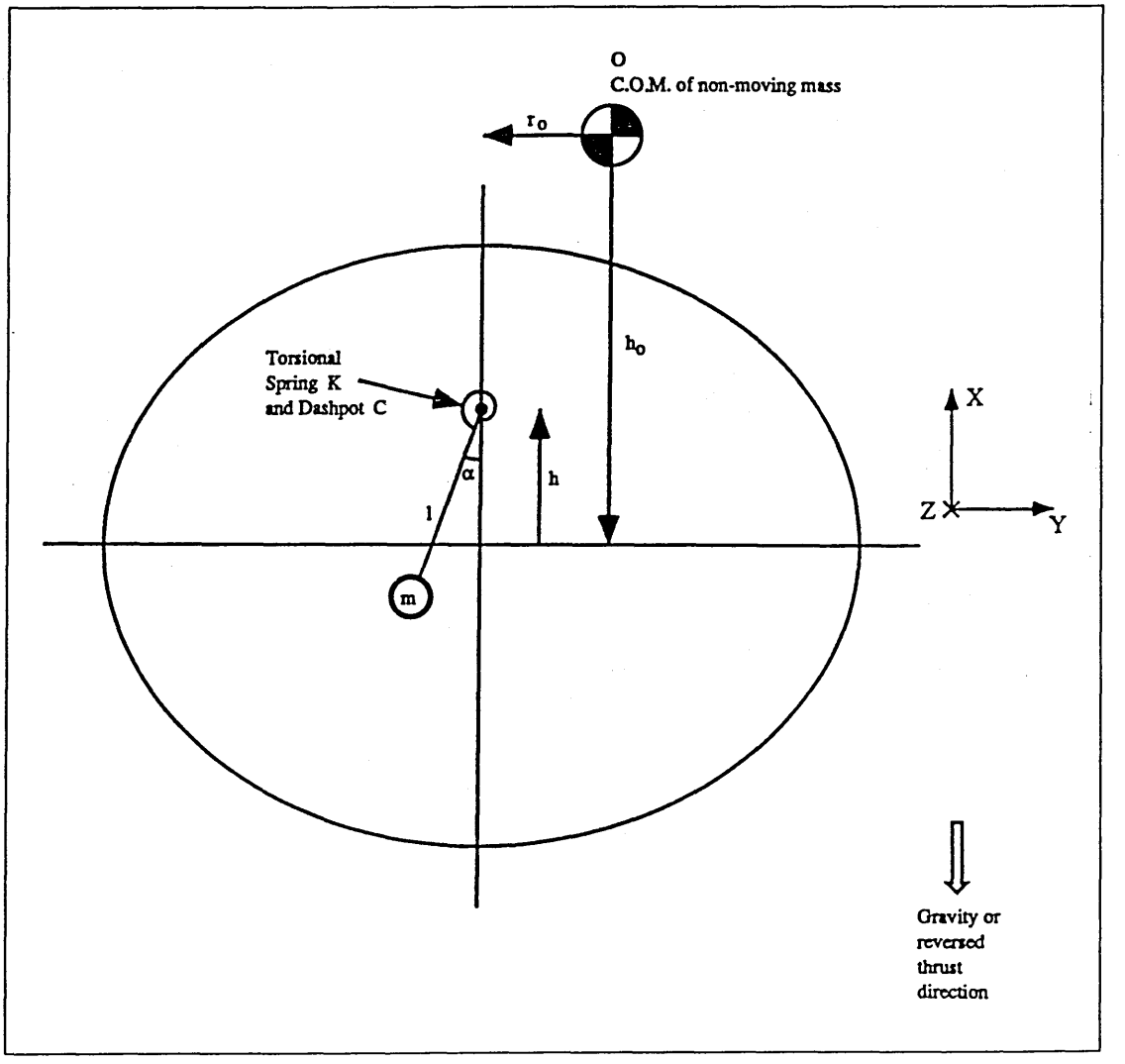

Fig. 4.11 : Pendulum model for +X manoeuvres

Mass of pendulum m	83.5 kg
Pendulum length l	0.111 m
Hinge height h	-0.094 m
Spring coefficient K	44.1 Nm/rad
Dashpot constant C	1.48 Nms/rad
Damping ratio ζ	0.11
Natural frequency ω	6.55 rad/s

Table 4.1 : Pendulum parameters for +X manoeuvres (from BAe model)

Using the values of the parameters in table 4.1, the transfer function shown in (4.36) becomes

$$\frac{\delta_f^2 s^2}{s^2 + 1.44s + 42.9} \tag{4.37}$$

Now all we need to do is calculate the inertia of the mode (the square of the coupling coefficient of the mode δ_f). This is obtained from

$$\delta_f^2 = Mr^2 \tag{4.38}$$

where M is the mass of the pendulum bob and r is the distance of the pendulum bob from the spacecraft centre of mass, and is given by

$$r = \left\{\begin{matrix}\text{Distance of} \\ \text{spacecraft COM} \\ \text{from separation plane}\end{matrix}\right\} - \left\{\left(\begin{matrix}\text{Distance of tank} \\ \text{centre from} \\ \text{separation plane}\end{matrix}\right) + \left(\begin{matrix}\text{Hinge} \\ \text{height } h\end{matrix}\right)\right\} + \left\{\begin{matrix}\text{Length of} \\ \text{pendulum } l\end{matrix}\right\} \tag{4.39}$$

For the values of the parameters given in the BAe model, r is then given by

$$r = 1.77 - (1.21 - 0.094) + 0.111 = 0.765 \text{ m} \tag{4.40}$$

The inertia of the mode is then given by

$$\delta_f^2 = 83.5 \text{ kg} \times (0.765 \text{ m})^2 = 48.9 \text{ Kgm}^2 \tag{4.41}$$

The transfer function for fuel sloshing given in (4.36) then becomes

$$\frac{48.9s^2}{s^2+1.44s+42.9} \tag{4.42}$$

Equation (4.42) gives a coarse approximation of the full BAe fuel sloshing model which incorporates cross-coupling between axes. However, it is sufficient for the purposes of the spacecraft model required for the work in this thesis. However, as the magnitude of the effect as represented in (4.42) is quite small, it was decided to increase the inertia of the mode by an arbitrary figure, so that a more challenging control problem was presented to the neural network controllers discussed in the next section. For this reason, the transfer function used to represent fuel sloshing in this work was arbitrarily selected to be

$$\frac{625s^2}{s^2+1.44s+42.9} \tag{4.43}$$

4.6 Summary of Spacecraft Model

In the previous sections we have gradually developed a spacecraft model for use in the investigation of neural network controllers for spacecraft control. At each stage, an additional spacecraft component was carefully derived and incorporated into the model, until finally a complete spacecraft model had been constructed, as illustrated in figure 4.10.

In considering each component of the spacecraft model, it was borne in mind that the Simulink simulation package was to be used, and that therefore, the derivation of a series of transfer functions would be the most appropriate way to go about constructing the spacecraft model.

Once all the transfer functions for the various components of the model have been obtained, they can quickly and efficiently be entered into the Simulink environment. The Simulink representation of the complete spacecraft model is shown in figure 4.12.

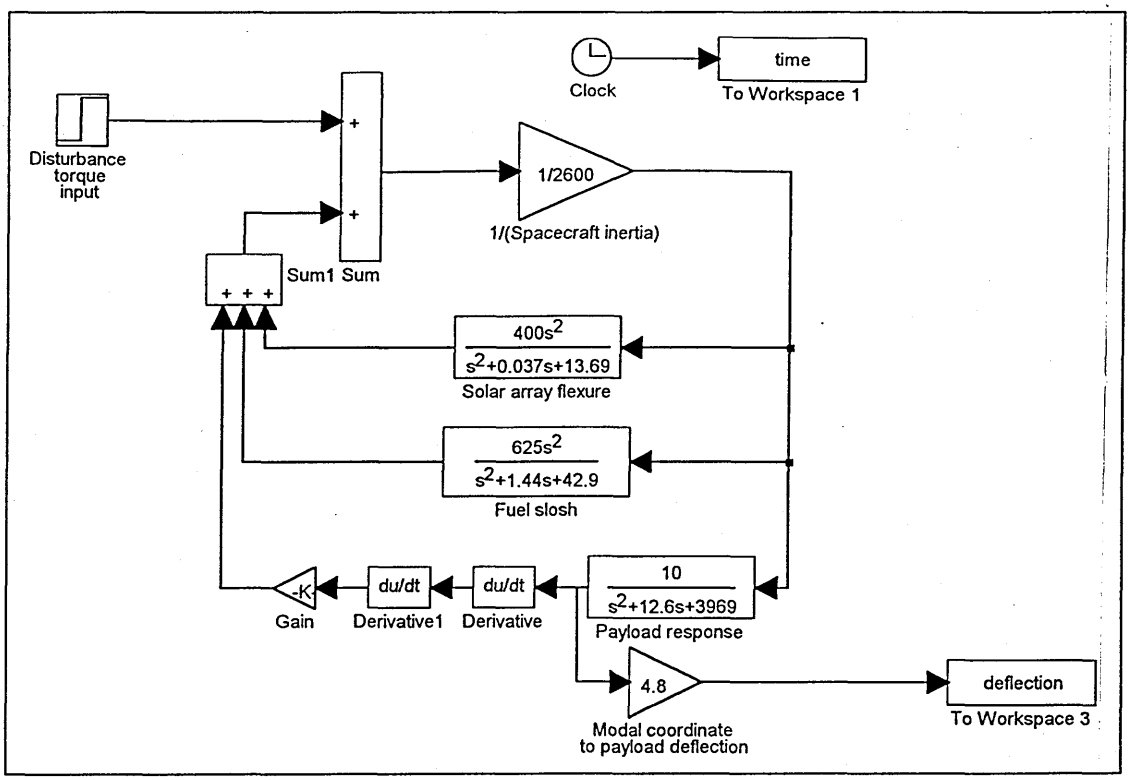

Fig. 4.12 : Simulink block diagram of the complete spacecraft model

Once represented in Simulink, the model can be simulated and monitored with ease using the Simulink graphical user interface. The model can also be rapidly edited, should changes to any of the parameters need to be made, and results of simulations plotted easily. The Simulink simulation package therefore served as a very powerful investigation tool for the work carried out in this thesis.

Although figure 4.12 illustrates the complete spacecraft model that was considered in this thesis, it does not include any details of how the spacecraft interfaces with the conventional control algorithms that would be used for coarse pointing control. A schematic diagram of the complete spacecraft control system is therefore shown in figure 4.13.

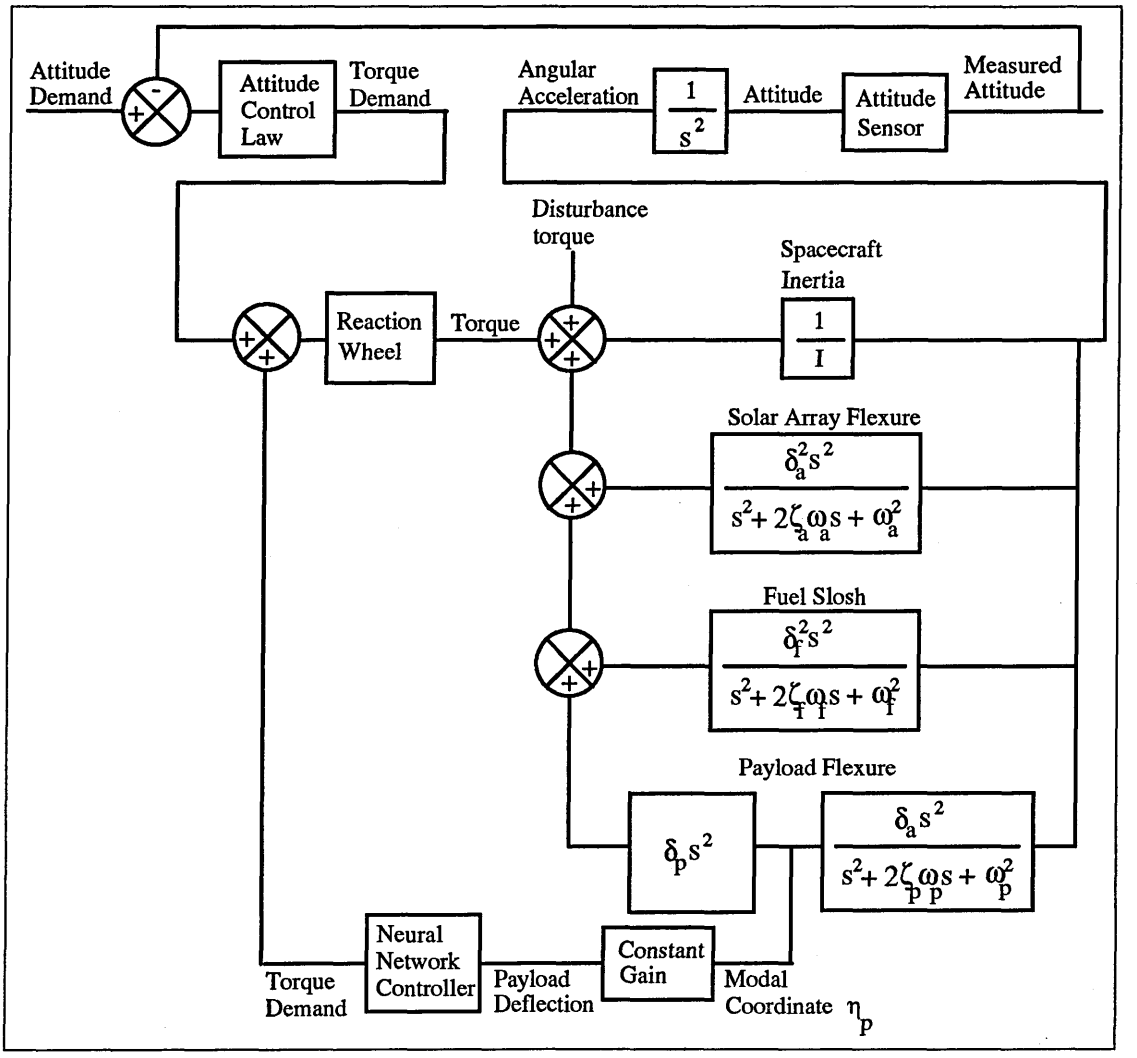

Fig. 4.13 : Schematic diagram of the complete spacecraft control system

Thus it can be seen that in this work, the objective was to construct a neural network controller for use in an inner control loop, with the outer control loop not being considered. It is envisaged that the inner control loop would act at a much higher frequency than the outer control loop that is used for coarse pointing of the spacecraft bus. It is therefore assumed that the two control loops can be considered independently as there is likely to be little interaction between the two.

Note that although figure 4.13 indicates that the inner block diagram contains a block representing the reaction wheel of the spacecraft, this component was not modelled in the work in this thesis. Torque demands created by the neural network controllers were assumed to create torques that are exerted on the spacecraft instantaneously.

This concludes the chapter on the spacecraft model that formed the subject of the investigations carried out in this thesis. In the following two chapters, neural network control techniques are applied to the control problem represented by this spacecraft model.

5 Direct Model Inversion

The first neural network control method to be applied to the spacecraft control problem detailed in the previous chapter was the Direct Model Inversion method. It was decided to investigate this method first as it is the simplest neural network control method available, and also the easiest to implement in the Simulink environment.

5.1 Controller Network Configuration

5.1.1 Choice of Training Vectors

The first step in producing a neural network controller using the Direct Model Inversion method is to generate a set of plant input and output vectors that can be used for network training. Although this seems straightforward enough, the generation of training vectors raises a number of important issues such as :

(i) What type of input should be fed to the plant for the purpose of generating training vectors ?

(ii) What output or outputs should be recorded ?

(iii) How many training vectors are required ?

There are no hard and fast rules that can be followed to answer these questions, but there are a number of guidelines that can be followed.

In answer to the first question, a good guideline is that the input that is supplied to the plant should be as representative as possible of the input that the plant is likely to receive when the neural network controller is required to be in operation. In this work, this means performing a simulation using a sinusoidal disturbance torque as represented by equation (4.1) as an input to the plant.

As for the second question, there is no real guideline that can be followed at all, other than to experiment and see what works best. Clearly, it would not be possible to train a

neural network to produce the current disturbance torque input to the spacecraft model when supplied with only the current payload deflection. This is because the network would not have been passed any information about the state of the plant at that particular time. It is therefore necessary to pass the neural network controller some further information, such as payload velocities, accelerations and/or details of the states of the other components of the spacecraft model. In this work, it was assumed that only payload positional information would be available to the controller, so only current and previous payload deflection measurements can be used as inputs to the neural network. This choice of inputs then raises the additional question of how many previous payload deflections should be used as inputs. One way of deciding is to initially use more inputs than necessary, and then to train the controller network a number of times, each time with one less input. This procedure is then halted at the point where reducing the number of network inputs further would result in too great a deterioration in network performance. However, this procedure was not followed in this work as reasonable results have already been obtained in the work carried out by [6] using the current payload deflection together with the last seven payload deflections as inputs to the controller network. Therefore, this configuration was selected for the work carried out in this thesis. The input and output vectors that were used to train the neural network controller are shown in figure 5.1.

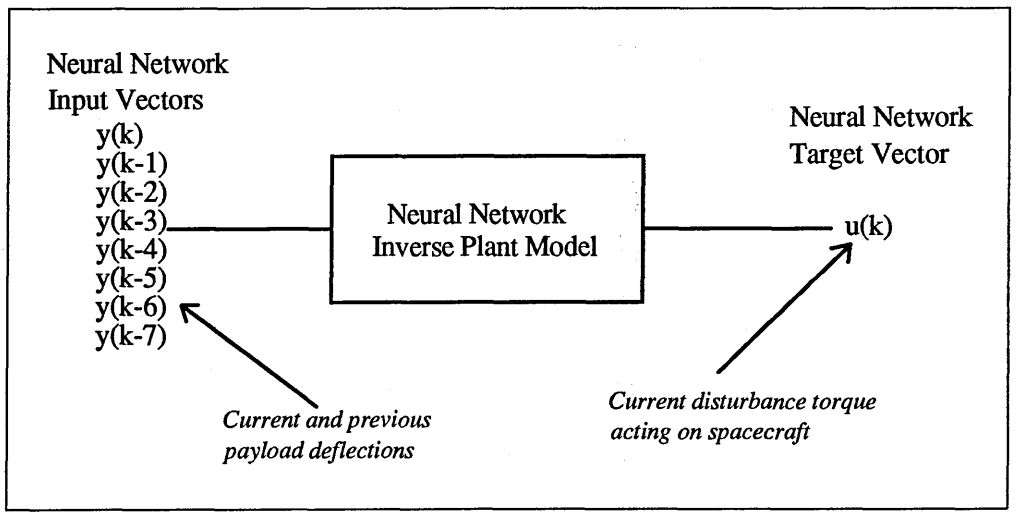

Fig. 5.1 : Direct Model inversion training vectors

Finally, consider the question of how many training vectors will be required. A good guideline is to choose a large enough number of vectors for the operating range of plant inputs and outputs to be fully spanned and sufficiently sampled. Alternatively, a larger number of training vectors than necessary can be initially used. This number is then

reduced, subject to the performance of the controller deteriorating below a certain acceptable level. In the work in this section, 400 training vectors were chosen, spanning a period of 2 seconds of simulation. This corresponds to plant inputs and outputs being sampled every 0.005s.

5.1.2 Training Vector Generation

The Simulink block diagram that was used to generate the training vectors is shown in figure 5.2. The "To Workspace" blocks are used to record the data that is generated during the simulations in matrices which can be accessed from within the Matlab workspace at the end of the simulation.

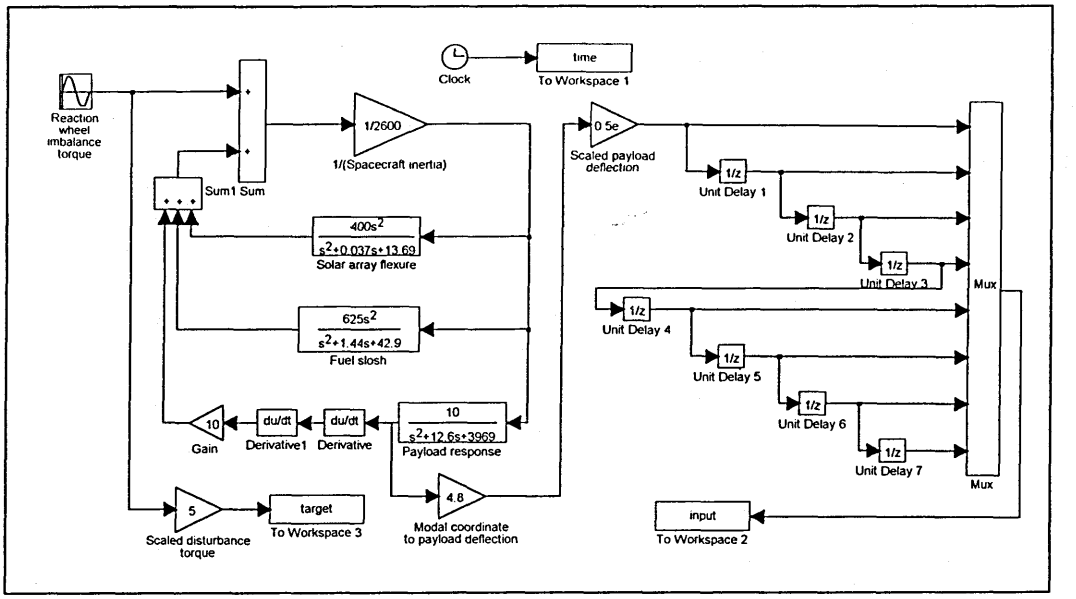

Fig. 5.2 : Simulink block diagram used to generate a set of training vectors

The Simulink block diagram shown in figure 5.2 was simulated for a period of 2 seconds using a Runge-Kutta 5^{th} order numerical integration method with a fixed step size of 0.005s. This resulted in plant inputs and outputs being sampled 401 times during the course of the simulation (although the "To Workspace" blocks were actually set to only record 400 elements).

The input to the plant was a sinusoidal disturbance torque of the form of equation (4.1) with a peak of 0.006 Nm and a frequency of 60 rad s^{-1}. At each of the time steps of the simulation, plant inputs and outputs were recorded in matrices called "target" and "input" respectively. Note that seven cascaded delay operators and a multiplex block were inserted before the "To Workspace" block labelled "input". By doing this, a matrix called "input" was constructed in the Matlab workspace that comprised the current and last seven payload deflections at each of the time steps of the simulation (i.e. a 8 by 400 matrix is formed). This was advantageous as it saved having to write the code that would be necessary to convert a 1 by 400 matrix (comprising current payload deflections at each of the time steps) into a matrix of current and delayed payload deflections that would be suitable for training the neural network controller.

A further point to note is that plant inputs were scaled up before being recorded in the Matlab workspace. This was done so that the magnitude of the largest output from the neural network controller (which was to be 1 since tansigmoidal neurons were to be used, see figure 2.9) would correspond to 0.2 Nm which is the largest torque that would be available from a SOHO reaction wheel. For the sake of consistency, plant outputs were also scaled up to lie in the region ±1. The scaling laws that were used for this purpose were :

$$\begin{pmatrix}\text{scaled torque used for}\\ \text{neural network training}\end{pmatrix} = 5 \times (\text{real torque value}) \qquad (5.1)$$

and

$$\begin{pmatrix}\text{scaled payload deflection used}\\ \text{for neural network training}\end{pmatrix} = (0.5 \times 10^{6}) \times (\text{real payload deflection}) \qquad (5.2)$$

Obviously, in using these scaled values, it must be remembered to scale up and down the inputs and outputs of the neural network controller once it has been trained and is in operation.

5.1.3 Training Procedure

Once a set of training vectors had been created, it was used to train a neural network controller. The network architecture that was adopted for the controller was that of a single tansigmoidal neuron with eight inputs (which were the current and last seven scaled plant deflections).

The weights and single bias of the network were first randomised in the region ±1. The network was then trained for 1,000 epochs (where one epoch represents one complete presentation of the training vector set to the network). The weights and bias of the network were updated using the backpropagation with momentum learning algorithm. The reduction in the sum-squared error of the network as training proceeded is shown in figure 5.3.

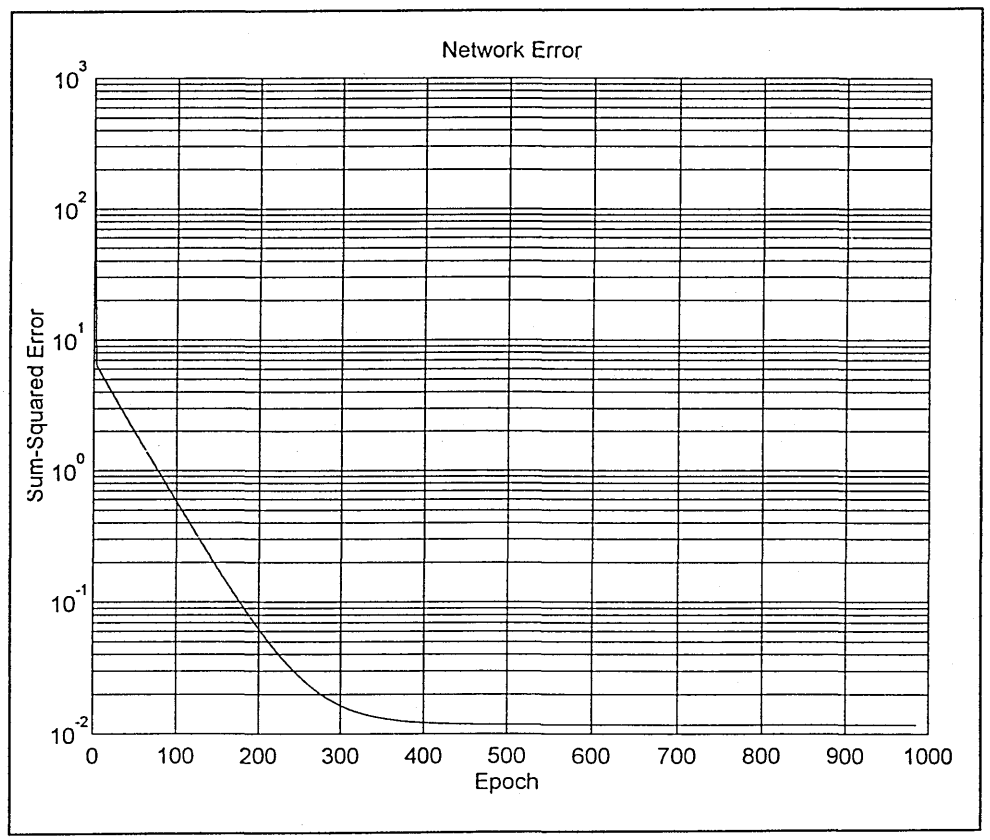

Fig. 5.3 : Neural network training record

The program that was used to train the neural network was written in "Matlab language" (Matlab language programs are basically scripts of Matlab commands, although conditional statements are also supported). A listing of the program that was used is

given for the sake of completeness in Appendix C. However, the main body of the program consists of the following commands :

$A = \text{tansig}(W \times P, B)$; calculate network output
$E = T - A$; calculate network error
$D = \text{deltatan}(A, E)$; calculate delta vector
$[dW, dB] = \text{learnbpm}(P, D, lr, mc, dW, dB)$; update weights and biases

In the above code, the "tansig" function returns the output of a network of tansigmoid neurons with input P, weights W and biases B. The error vector E is calculated by subtracting the network output A from the target output T . The delta vector is then calculated for the network by calling the "deltatan" function. Finally the "learnbpm" (learn-backpropagation-momentum) function is called to calculate the changes that are to be made to the weights and biases of the network from the input vector P, the delta vector D, the learning rate lr, the momentum constant mc, and the previous changes to the network weights and biases. In the work in this section a learning rate of 0.05 and a momentum constant of 0.95 were chosen.

The above code is looped until either the network error drops below a predetermined level or the maximum number of epochs is reached.

5.1.4 Representation in Simulink

Once a neural network controller has been trained, it can be interfaced with the Simulink environment using a "Matlab Function" block.

For a single layer network comprising a single tansigmoidal neuron, the Matlab command that gives the network output is

$$\text{tansig}(W \times P + B) \tag{5.3}$$

where W and B are the matrices in the Matlab workspace that store the weights and bias of the neuron. Once the controller network is fully trained, it can therefore be used in the Simulink environment using a "Matlab Function" block that calls the command given by equation (5.3) at each time step of the simulation.

Figure 5.4 shows the Simulink block diagram that was used for controller simulations. note that the "Matlab Function" block that represents the trained neural network controller was simply inserted in the negative feedback path of the plant.

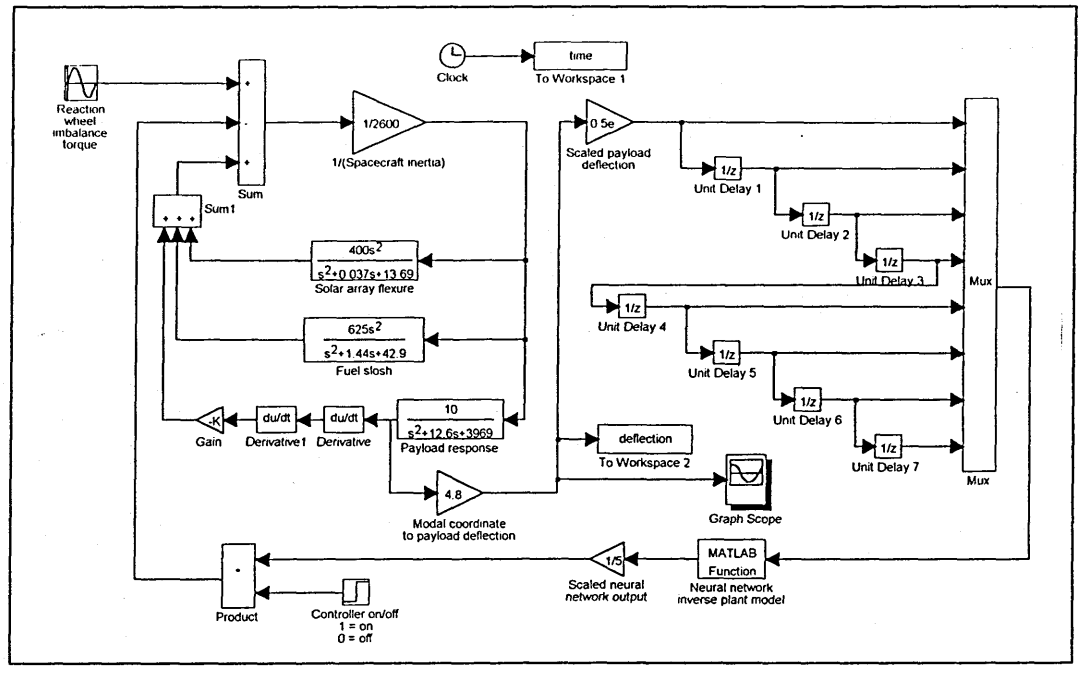

Fig. 5.4 : Simulink block diagram used for Direct Model Inversion simulations

5.2 Controller Operation

The Simulink block diagram shown in figure 5.4 was simulated for a period of 25 seconds using a Runge-Kutta 5th order numerical integration routine. The disturbance torque input that was used for the simulation was chosen to be the same as that used for training the neural network controller. During the simulations the controller was initially left switched off, but was switched on after 11 seconds. Once in operation, the controller provided control signals every 0.005s.

5.2.1 Controller Performance under Standard Conditions

The response of the payload to the disturbance torque with the neural network controller being switched off and then on after 11 seconds is shown in figure 5.5

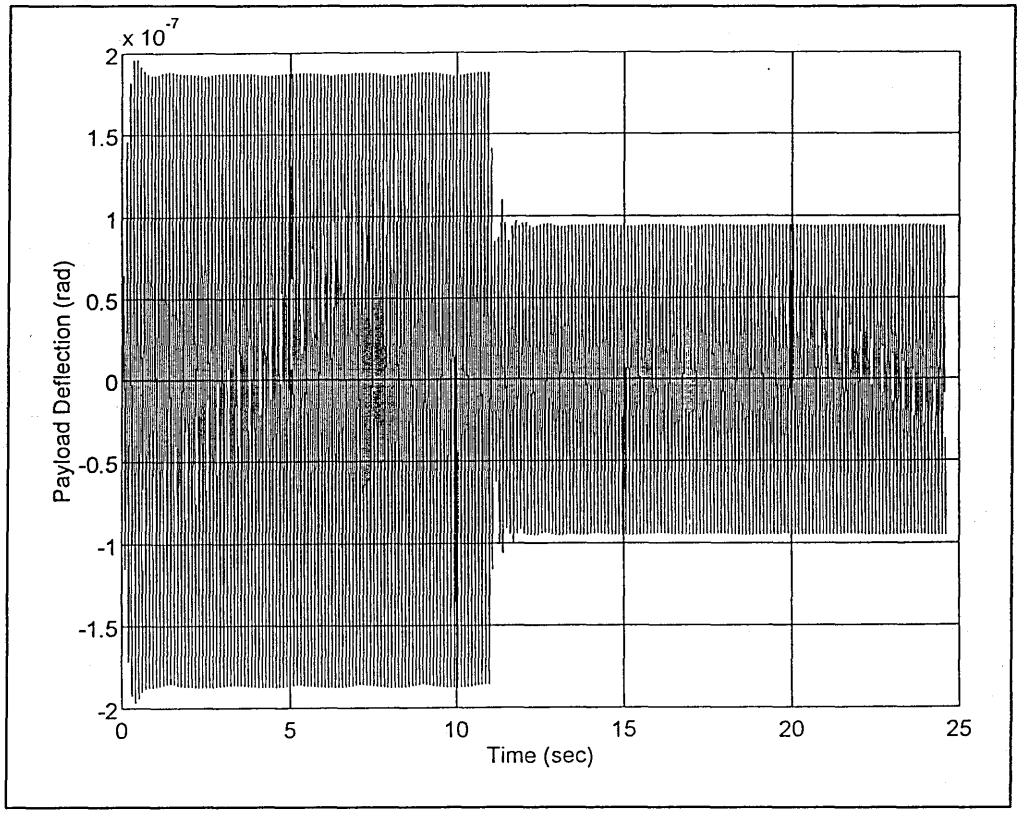

Fig. 5.5 : Controller performance (wheel speed = 60 rad s^{-1})

As can be seen in figure 5.5, when the neural network controller is switched on, payload deflections are reduced by a factor of one half, as predicted by equation (3.2). The Direct Model Inversion method has therefore been successfully used to train a neural network to model the inverse dynamics of the plant.

5.2.2 Controller Performance under Non-Standard Conditions

Although in the previous section, it was shown how a neural network controller was successfully trained to reduce payload deflections, there was no discussion of how well the controller can perform when differing disturbance torque inputs are fed to the plant.

After all, figure 5.5 just represents the response of the payload when the same disturbance torque that was used to train the neural network controller is input into the plant. A good test of the usefulness of the neural network controller would therefore be to repeat the simulations using disturbance torque inputs corresponding to different reaction wheel speeds.

Figures 5.6 to 5.8 show the performance of the neural network controller when disturbance torques corresponding to differing reaction wheel speeds were fed to the plant. In the three figures, reaction wheel speeds of 50 rad s^{-1}, 40 rad s^{-1} , and 30 rad s^{-1} were used respectively.

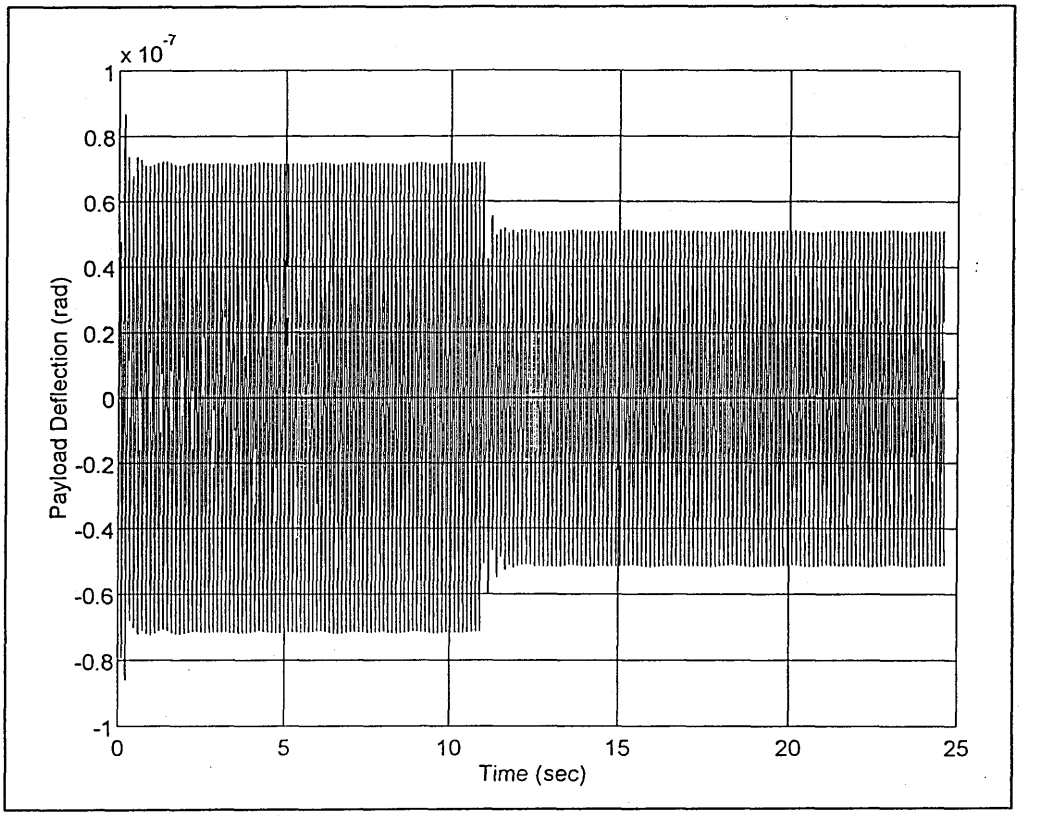

Fig. 5.6 : Controller performance (wheel speed = 50 rad s^{-1})

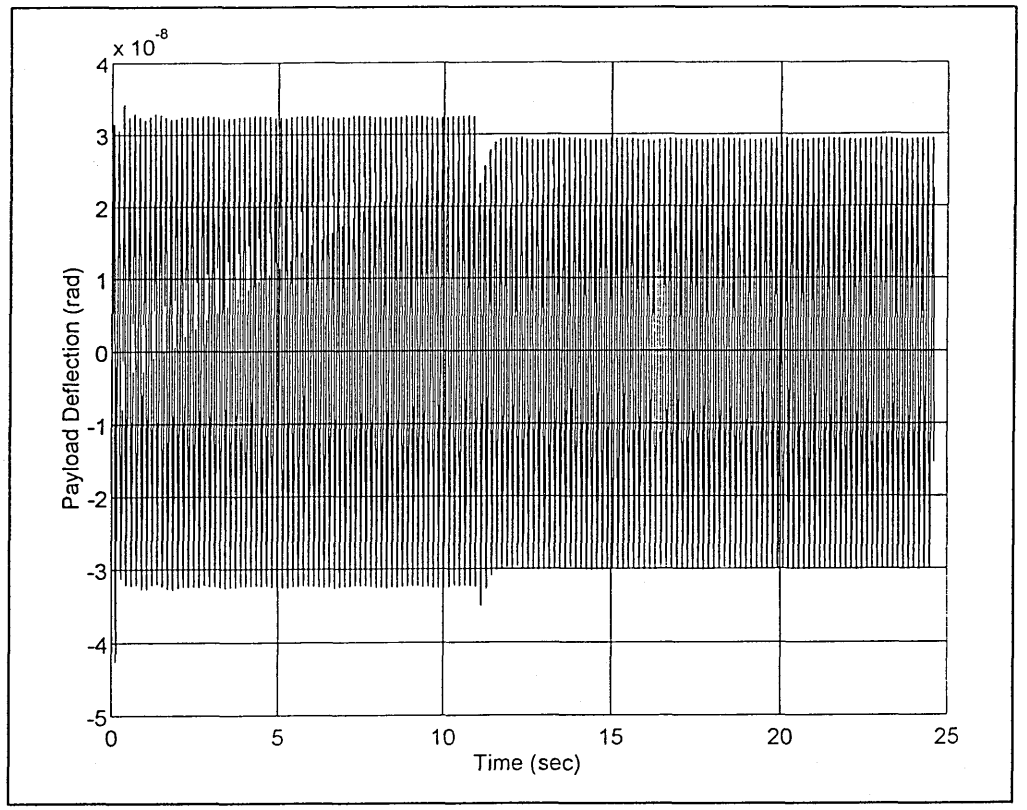

Fig. 5.7 : Controller performance (wheel speed = 40 rad s^{-1})

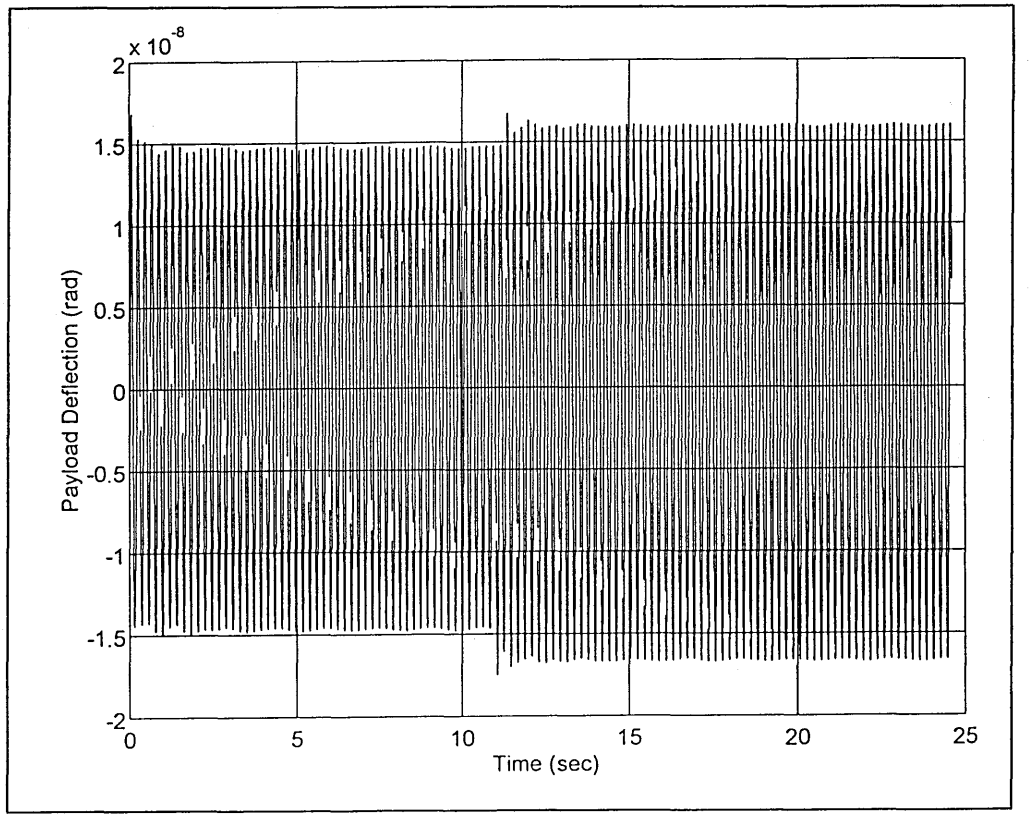

Fig. 5.8 : Controller performance (wheel speed = 30 rad s^{-1})

As can be seen, the neural network controller is still capable of reducing payload disturbances when reaction wheel speeds of 50 rad s^{-1} and 40 rad s^{-1} are used (which are still close to the reaction wheel speed of 60 rad s^{-1} that was used for training vector generation). However, the performance of the controller drops off rapidly with decreasing reaction wheel speed, and once the wheel speed drops to 30 rad s^{-1}, turning on the controller after 11 seconds actually causes payload disturbances to increase.

Finally, to test the performance of the controller under the conditions of unexpected changes in the plant dynamics, a simulation was carried out where the inertia of the spacecraft decreased by 10% after the first 5 seconds. As before, the controller was turned on after 11 seconds. The result of this simulation is shown in figure 5.9.

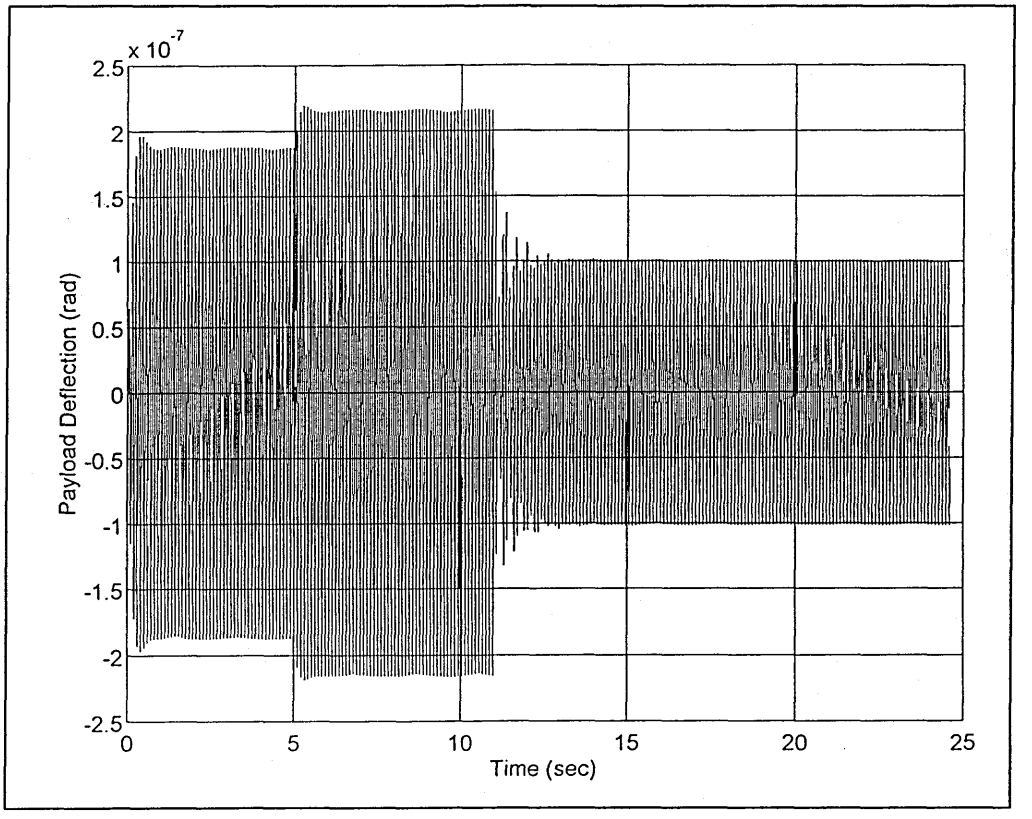

Fig. 5.9 : Controller performance (reduction in spacecraft inertia after 5 seconds)

As can be seen in figure 5.9, the unexpected 10% change in spacecraft inertia after the first 5 seconds of the simulation does not drastically change the dynamics of the plant, and the controller is still able to reduce payload disturbances by approximately one half.

5.3 Conclusions

In this chapter, the Direct Model Inversion method was used to train a neural network controller to model the inverse dynamics of the spacecraft plant model developed in Chapter 4. It was shown that the neural network controller that was produced using this method was successful in reducing payload deflections by approximately one half, in accordance with theoretical predictions. It was also shown that the controller could generalise to a certain extent when presented with different plant conditions to those experienced during training, although performance decreased rapidly with decreasing frequency of the disturbance torque inputs.

Despite the seemingly successful results obtained using the Direct Model Inversion method, a payload disturbance rejection factor of one half does not represent a major achievement for the neural network controller when compared to the performance of traditional control methods, such as using a simple gain term in the feedback path. The result highlights the shortcomings of using a pure plant inverse for closed loop control, and hence highlights the inadequacies of the Direct Model Inversion method.

The main problem with the Direct Model Inversion method is that it does not allow any information about the desired operation of the plant to be built into the neural network controller. In addition, the method does not take into consideration the possibility that more than one input to the plant could produce the same plant output.

The drawbacks of the Direct Model Inversion method can be overcome to a certain extent by using the Indirect Model Inversion technique. The application of this technique to the spacecraft plant model is discussed in detail in the following chapter.

6 Indirect Model Inversion

The Indirect Model Inversion method is quite different to the Direct Model Inversion method discussed in the previous section. The method is also much more difficult to implement, particularly in the Simulink environment, and was therefore investigated after satisfactory results had been obtained using the Direct Model Inversion method.

6.1 Forward Model Configuration

6.1.1 Choice of Training Vectors

One of the important differences between the Direct Model Inversion and the Indirect Model Inversion methods is that the latter method makes use of two neural networks. The first network is used to model the forward dynamics of the plant to be controlled, and is trained off-line using training vectors generated from the plant. The second network is the controller network, and is trained on-line in accordance with the performance of the network as a controller.

The first step to take when using the Indirect Model Inversion method is to choose an appropriate architecture for the neural network that is to model the forward dynamics of the plant.

The forward dynamics of linear plants can be well represented using an Auto-Regressive Moving Average (ARMA) model. In this scheme, the next output of the plant can be calculated from the sum of weighted previous plant inputs and outputs. Mathematically, the ARMA model is represented as

$$y_{k+1} = \sum_{i=0}^{n} a_i y_{k-1} + \sum_{i=0}^{n} b_i u_{k-i} \tag{6.1}$$

Equation (6.1) can be easily represented by a neural network comprising a single linear neuron, where a and b are the weights of the inputs to the neuron. The only question to ask now is how many past plant inputs and outputs must be used in the ARMA model. Again, a good way of answering this question is to use more inputs and outputs than are

necessary, and then to gradually reduce this number until further reductions would result in the performance of the forward model falling below a predetermined level. However, good results have been obtained in [6] using the current and previous three plant inputs and outputs, so this configuration was adopted again here.

6.1.2 Training Vector Generation

The Simulink block diagram that was used to generate training vectors for the forward model neural network is shown in figure 6.1. A breakdown of the Simulink sub-block labelled "Delayed plant inputs and outputs" is shown in figure 6.2.

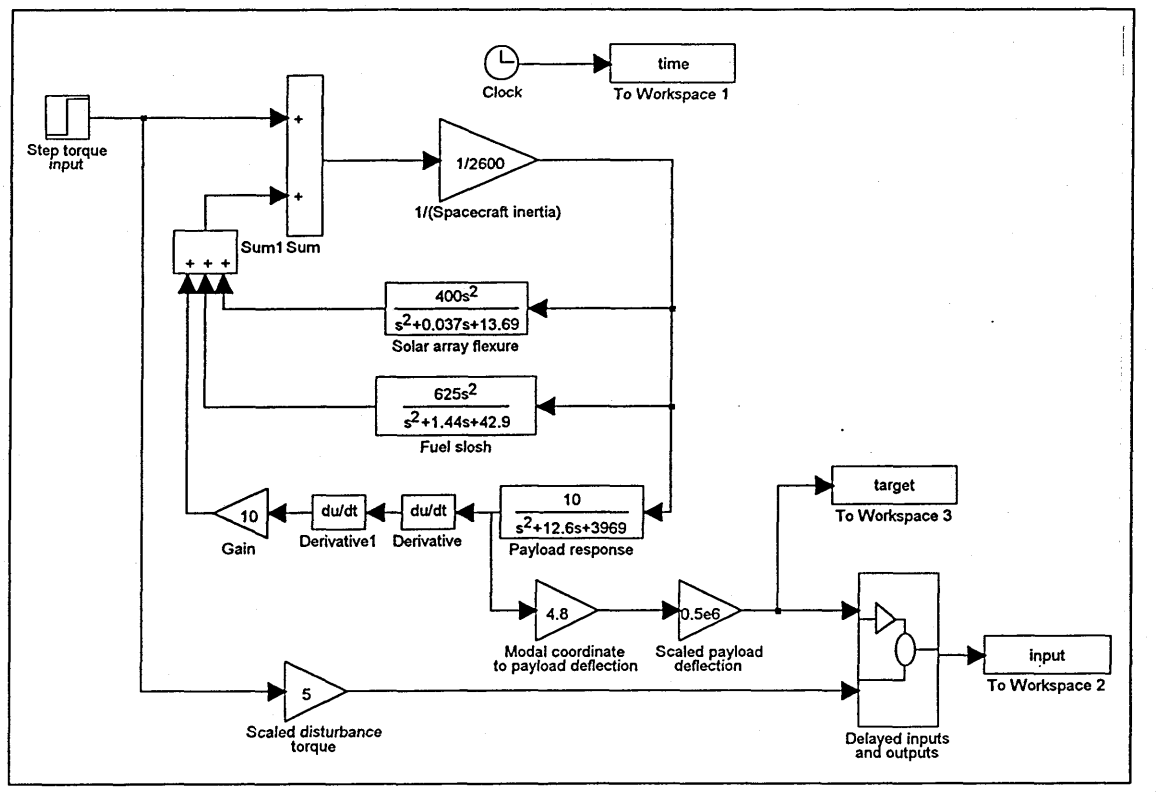

Fig. 6.1 : Generation of training vectors for the forward model neural network

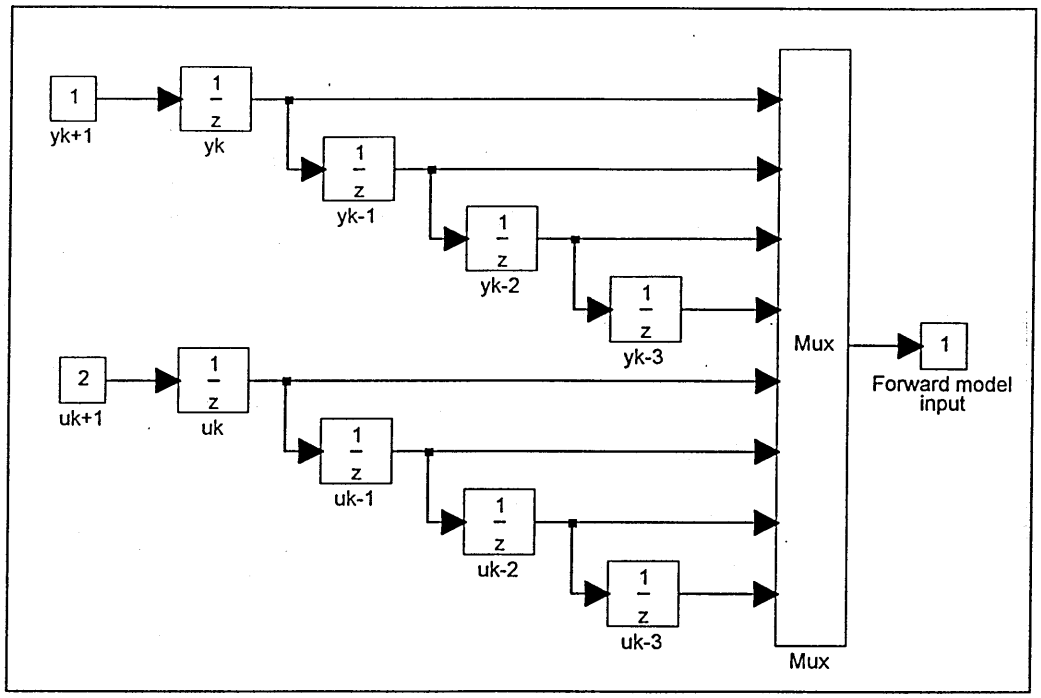

Fig. 6.2 : Simulink sub-block used to generate delayed plant inputs and outputs

The Simulink block diagram shown in figure 6.1 was simulated for 2 seconds using a Runge-Kutta 5th order numerical integration routine with a fixed step size of 0.005s. The input to the plant was a step torque of 0.2 Nm (which is the maximum torque that is available from the reaction wheel of the spacecraft). At the end of the simulation, the training vectors were stored as matrices in the Matlab workspace. Once again, the use of the appropriate number of delay operators (as illustrated in figure 6.2) saved having to write a Matlab language program to construct the appropriate training vectors from matrices containing the single inputs and outputs at each of the time steps.

6.1.3 Training Procedure

Once the training vectors had been generated, they were used to train the neural network that was to represent the ARMA model. The Matlab code needed to train a single linear neuron with input matrix P and a weight matrix W is :

$A = \text{purelin}(W \times P)$; calculate network output

$E = T - A$; calculate network error

$dW = \text{learnwh}(P, E, lr)$; update weights

In the above code the "purelin" function calculates the output of a linear neuron. E is the error vector, calculated from the difference between the actual output A of the neuron and the target output T. The "learnwh" (Learn-Widrow-Hoff) function is used to calculate the matrix of weight changes dW for the linear neuron from the neuron input P, the error vector E, and the learning rate lr using the Widrow-Hoff rule. The value of the learning rate must be carefully chosen as too small a learning rate results in excessively large training times. On the other hand, too large a learning rate results in the network continually overshooting the minimum in the error surface.

The Widrow-Hoff learning rule is chosen for linear networks as the rule represents a steepest gradient descent procedure, and networks of linear neurons have parabolic error surfaces. A simple steepest gradient descent procedure is therefore guaranteed to find the minimum in the error surface.

Unlike most neural network architectures, networks of linear neurons can be designed directly without having to use the Widrow-Hoff learning rule as long as all the input and target vectors are known. The Matlab function that calculates the minimum error solution for a network of linear neurons is called "solvelin". Thus the code detailed above was not actually required, and was replaced instead with the command

$$W = \text{solvelin}(P,T) \tag{6.2}$$

where P and T were the matrices holding the input and target vectors respectively. The function of the solvelin routine does not involve any complex computation. The function simply finds the weight matrix W that solves the equation

$$WP = T \tag{6.3}$$

Once trained, the forward model network was tested by supplying a step torque to the input of the neural network. The step torque was chosen to be the same as that used to generate training vectors for the forward model from the plant. The output of the forward model network was then compared to the response of the actual plant when the same step torque input is used. The Simulink block diagram that was used to carry out this test is shown in figure 6.3. The responses of both the forward model neural network and the actual plant to the step torque are plotted together in figure 6.4.

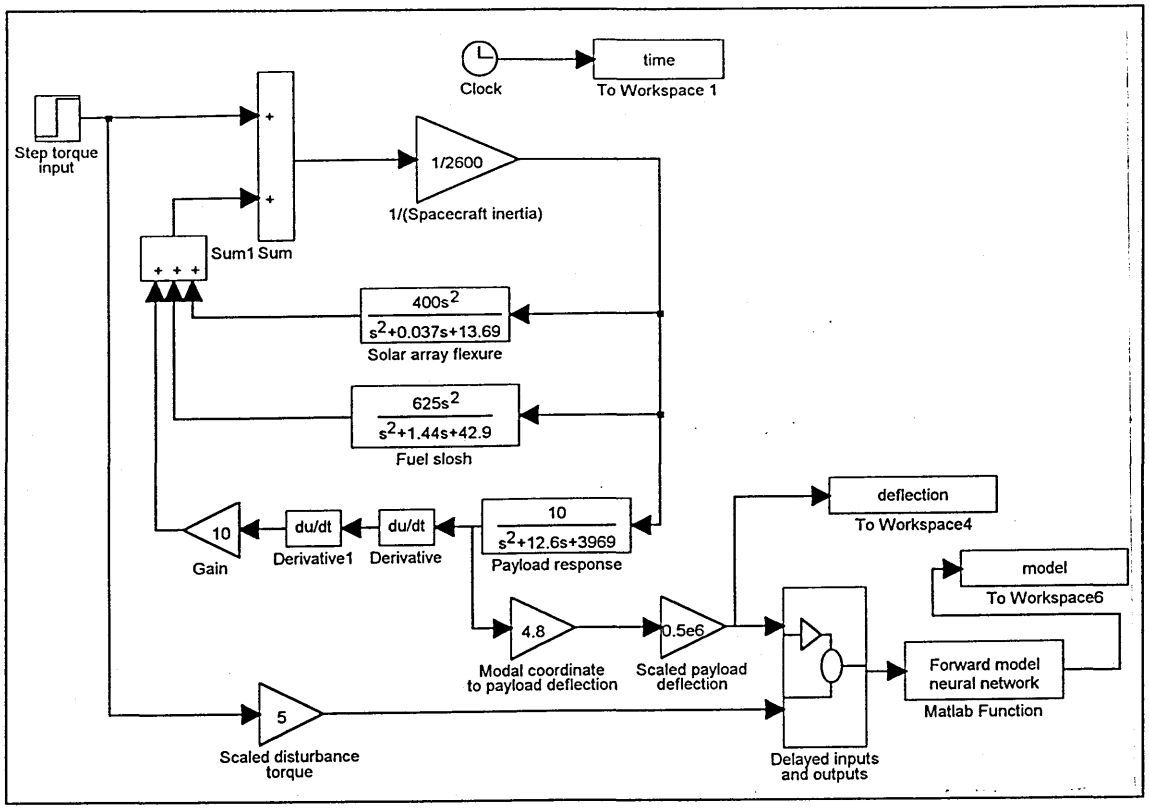

Fig. 6.3 : Simulink block diagram used to test the forward model neural network

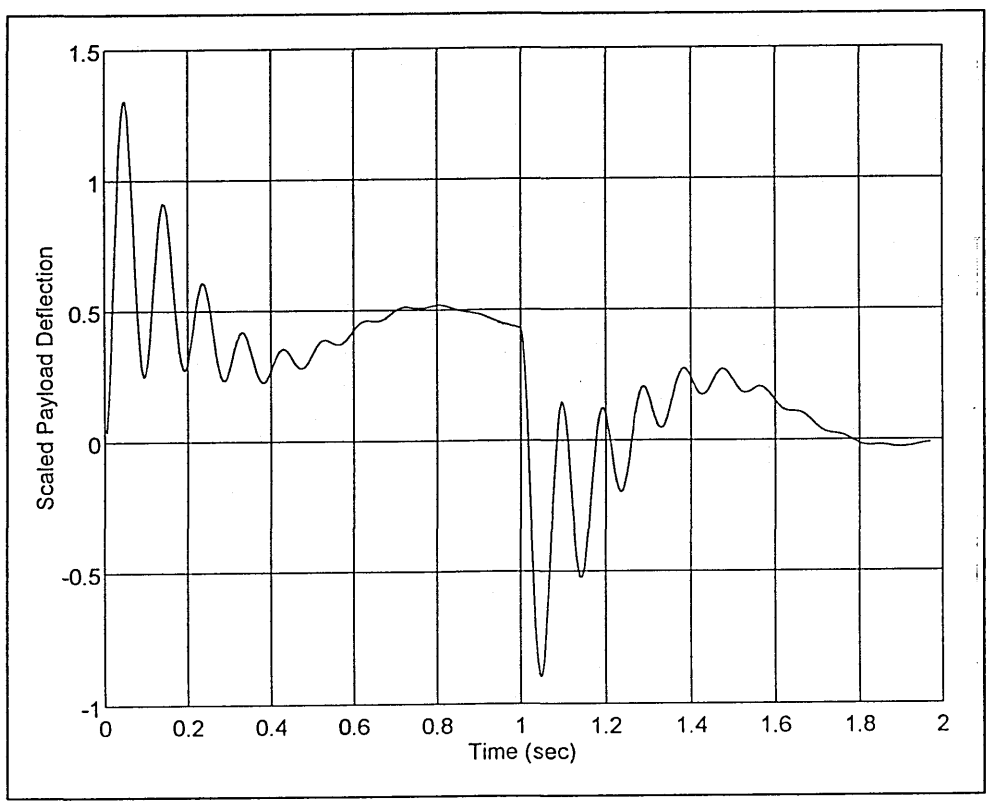

Fig. 6.4 : Plant and forward model neural network responses to a step torque input

As can be seen from figure 6.4, the responses of the plant and the forward model are virtually identical (the two plots almost completely overlap) indicating that the neural network models the forward dynamics of the plant well when presented with the same step torque that was used for training.

In order to test how well the neural network models the forward dynamics of the plant in the presence of "unseen" inputs, its performance was tested using a band-limited white noise input. Figure 6.5 shows the response of the neural network to a white noise input, along with the actual plant response.

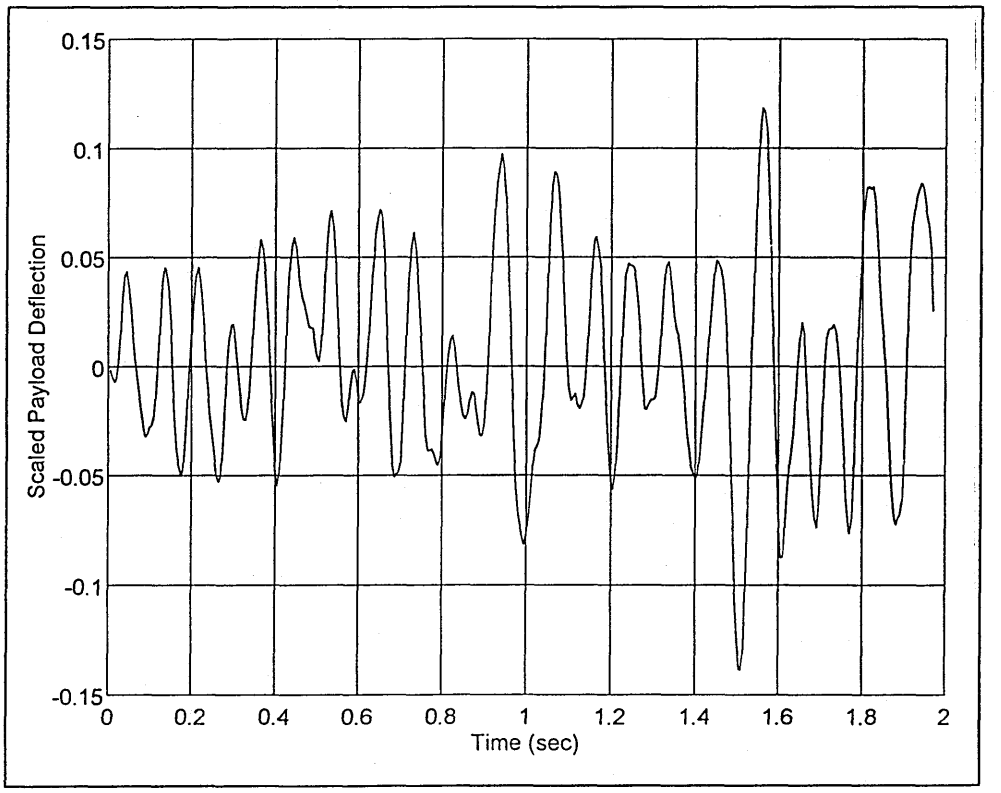

Fig. 6.5 : Plant and forward model neural network responses to a white noise input

Again, the two plots are virtually identical, indicating that the network is likely to give a good description of the forward dynamics of the plant in the presence of a variety of inputs.

6.1.4 Representation in Simulink

The forward model neural network served two purposes in the Indirect Model Inversion simulations carried out in this thesis. The first purpose was to calculate the next output of the plant (the next plant error) from current and previous plant inputs and outputs. The second purpose was to map the plant error from "plant output space" to plant input space, so that it could be used to update the weights and biases of the controller network. Once the forward model neural network had been obtained, these two roles could be implemented in Simulink using appropriate "Matlab Function" blocks. The Simulink block diagram representing this configuration is shown in figure 6.6.

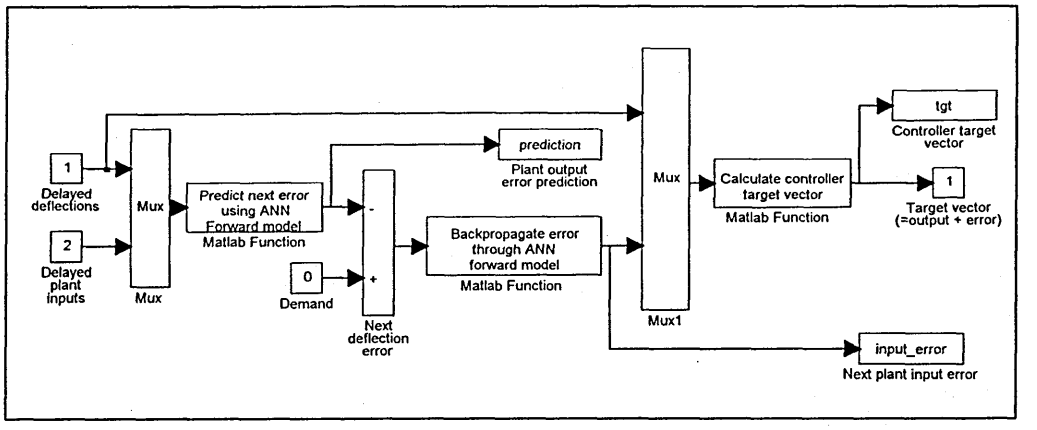

Fig. 6.6 : Simulink representation of the forward model neural network

The first of the Matlab Function blocks in figure 6.6 simply multiplies its input (comprising the current and last three plant inputs and outputs) with the weight matrix of the single neuron in the forward model network. The output of the forward model is then a prediction of the next output of the plant. This predicted output is then subtracted from the plant demand (which is zero) to obtain a prediction of the next plant error. before this error prediction can be used to update the weights of the controller network, it must be mapped from plant output space to plant input space by backpropagating it through the forward model.

The output of the forward model neural network is given by

$$y_{k+1} = w_1 y_k + w_2 y_{k-1} + w_3 y_{k-2} + w_4 y_{k-3} + w_5 u_k + w_6 u_{k-1} + w_7 u_{k-2} + w_8 u_{k-3} \qquad (6.4)$$

where u and y are the inputs and outputs of the plant respectively, and w_n is the weight of the connection of the n^{th} input to the neuron. Thus the relationship between the error in the current plant input and the error in the next plant output is given by

$$\delta u_k = \frac{\delta y_{k+1}}{w_5} \tag{6.5}$$

The error in the current plant input can therefore be simply calculated by determining the next plant deflection (plant error) and dividing it by the single network weight w_5. This then, is the function of the second of the Matlab Function blocks shown in figure 6.6.

The third Matlab Function block in figure 6.6 is simply used to construct a target vector for the controller network from the controller output vector and the error vector.

6.2 Controller Network Configuration

6.2.1 Network Architecture

As discussed in Chapter 3, a network comprising a hidden layer of tansigmoidal neurons and an output layer of linear neurons is capable of mapping any arbitrary function to any given precision provided a suitable number of neurons are chosen for the hidden layer and a suitable training procedure is adopted.

In view of the above, the architecture that was chosen for the neural network controller in this work was that of a hidden layer of tansigmoidal neurons and an output layer comprising a single linear neuron. The number of tansigmoidal neurons used in the input layer was varied during simulations between 1 and 10 in an attempt to find an optimum configuration for the controller network.

6.2.2 Representation in Simulink

Although it is easy to decide upon a suitable architecture for the neural network controller, it is not obvious how the network should be interfaced with the Simulink environment for Indirect Model Inversion simulations. In this section, the problems that arise in representing the Indirect Model Inversion controller in Simulink are discussed and the solution that was adopted to solve these problems is presented.

Although the controller used in the Direct Model Inversion method was simple to represent in Simulink, the controller used for Indirect Model Inversion is much harder to implement. This is because unlike the Direct Model Inversion method, the Indirect Model Inversion method is an on-line method, and as such, the weights and biases of the neural network controller must be updated during the course of a simulation. This presents problems as it means that the controller neural network cannot now be simply represented in Simulink by using another Matlab Function block since weights that are stored in the Matlab workspace cannot be updated while a simulation is in progress. Instead, some method of storing and updating weights and biases within Simulink must be found.

One way of dealing with this problem is to represent the controller network in Simulink using an S-Function block. This is a Simulink block that passes values to and receives values from a specially written program at specified times during a simulation. The program is written in Matlab language, but must follow a precise format so that the Simulink simulation can properly interact with it.

Figure 6.7 shows the Simulink block diagram that was used for the Indirect Model Inversion simulations carried out in this work. In the figure, the sub-block labelled "plant" consists of the spacecraft model developed in chapter 4, and the sub-block labelled "Neural network forward model" consists of the block diagram shown in figure 6.6. The sub-block labelled "neural network controller" is shown in figure 6.8

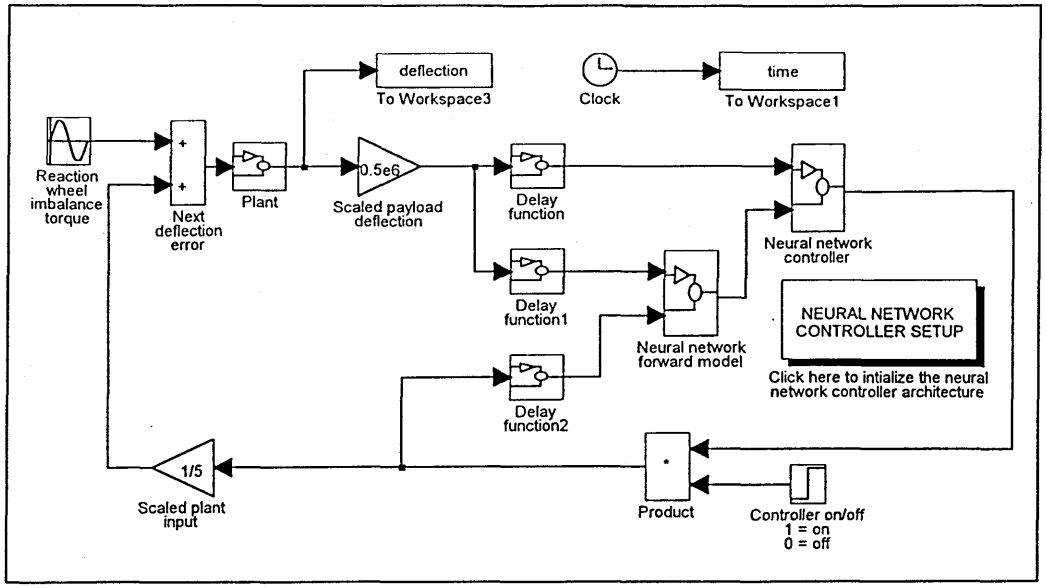

Fig. 6.7 : Simulink block diagram used for Indirect Model Inversion simulations

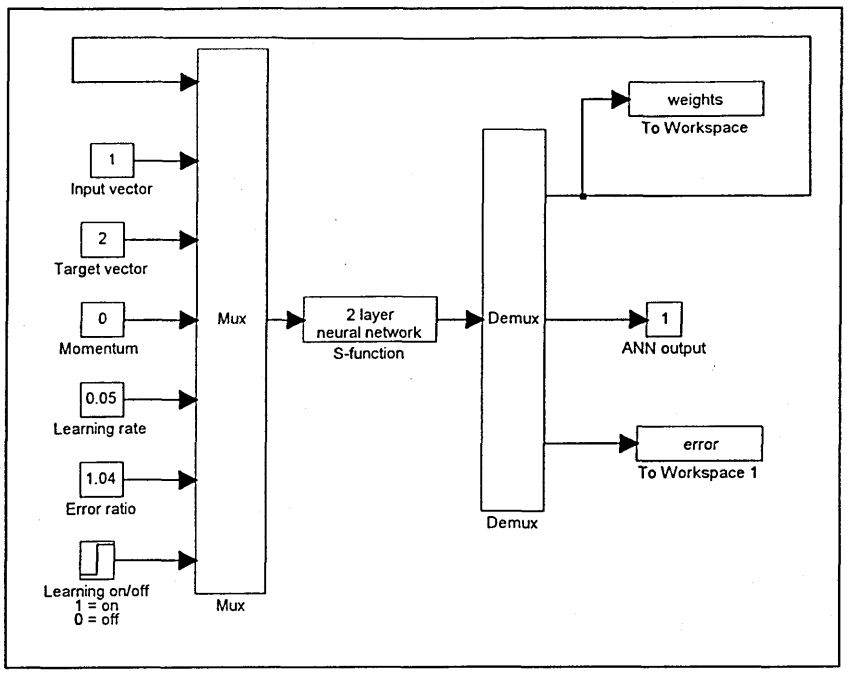

Fig. 6.8 : Simulink block representation of the neural network controller

Figure 6.8 shows how the S-Function block was used to represent the neural network controller in Simulink. To allow the greatest possible flexibility, the S-Function was designed in a way that allowed a number of parameters associated with the learning algorithm of the network to be adjusted during the simulations.

The operation of the S-Function can be described as follows : At the beginning of a simulation, the network weights and biases (which must be initially available in the

Matlab workspace) are passed to the S-Function along with the input and target vectors and the various parameters used by the network learning algorithm. The S-Function then calculates the network output and from that, the network error. This error is then used in conjunction with the parameters of the learning algorithm to calculate the new weights and biases of the network. Finally the new weights and biases, the network output and the network error are passed back from the S-Function to the Simulink simulation.

An important point to note about the operation of the S-Function is that the weights and biases of the controller network are passed in a feedback loop from the output to the input of the S-Function ready for use at the next integration time step. This is the only way in which the latest weights and biases of the controller network can be stored in the Simulink environment.

Finally, when the end of the simulation is reached, the weights and biases of the controller network are stored in the Matlab workspace. However, the Simulink block diagram representing the neural network was designed so that these stored weights and biases could easily be "re-loaded" by the network and used again in future simulations.

A major problem that arises with the operation of the S-Function discussed so far is that an S-Function can only receive and return matrices with a single row, and all the inputs and outputs of the S-Function used here are matrices of varying dimensions. This means that the S-Function must initially decode its input matrix into the weights, biases, input vector, target vector, and individual learning parameters before it can make use of them. Once the S-Function has completed its calculations it must then re-code the weights, biases, network output and network error into a single matrix of order (1 by n) ready for returning to Simulink. This greatly increases the complexity of the S-Function code that must be written to represent the neural network controller, particularly in view of the fact that networks with a various numbers of hidden neurons were to be tested. Thus the S-Function code had to be designed with sufficient flexibility to cope with a varying numbers of network weights and biases.

To achieve the flexibility that was required from the S-Function representing the neural network controller, it was necessary to also write a controller network set-up program that had to be run prior to carrying out a simulation. The program could be run from the Matlab command line or alternatively by clicking on a "neural network controller set-up" icon from within Simulink. The set-up program allowed the user to specify the number of hidden neurons in the controller network along with the number of inputs and the

delay time that was required between sampled inputs. The program also enabled the user to randomise the initial weights and biases of the controller network, if this was required.

In summary, the method that was used to represent the neural network controller in Simulink allowed controllers of varying architectures to be constructed quickly and easily. The method also allowed great flexibility in that learning parameters could be varied easily on-line, and that previously stored weights and biases could be re-loaded into the controller. However, one disadvantage with the method is that when networks using large numbers of neurons are used, the s-function must decode and re-code a considerable number of weights and biases. This drastically slows down the speed of the simulations.

For the sake of completeness, listings of the S-Function program and the controller set-up program that were used in this work are given in Appendix A.

6.3 Controller Operation

The Simulink block diagram shown in figure 6.7 was used for all the Indirect Model Inversion simulations carried out in this work, and in each case, a Runge-Kutta 5th order numerical integration routine with a fixed step size of 0.005s was used.

At the beginning of each of the simulations, the architecture of the controller network was initialised using the controller set-up program, and the initial weights and biases of the network were randomised. In each of the simulations discussed in sections 6.3.1 and 6.3.2, the architecture of the controller network comprised a single tansigmoidal neuron in the hidden (input) layer and a linear neuron in the output layer. In section 6.3.3 the results obtained using a larger number of tansigmoidal neurons in the output layer are discussed.

Each of the simulations was carried out for a period of 22 seconds. The main reason for choosing this length of time was that longer simulations caused the Matlab and Simulink programs to crash every time (this problem is discussed in section 6.3.4).

In each of the simulations, the neural network controller was switched on and set to adapt itself on-line for the first 20 seconds of the simulation. The controller was then

switched off so that the response of the uncontrolled plant could be observed for comparison.

6.3.1 Controller Performance under Standard Conditions

Figure 6.9 shows the performance of the neural network controller when a disturbance torque corresponding to a reaction wheel speed of 60 rad s^{-1} is fed to the input of the plant model. In this simulation a zero momentum constant was chosen for the learning algorithm so that network weight changes were based solely on error gradient and not at all on previous weight changes. A learning rate of 0.05 was arbitrarily selected.

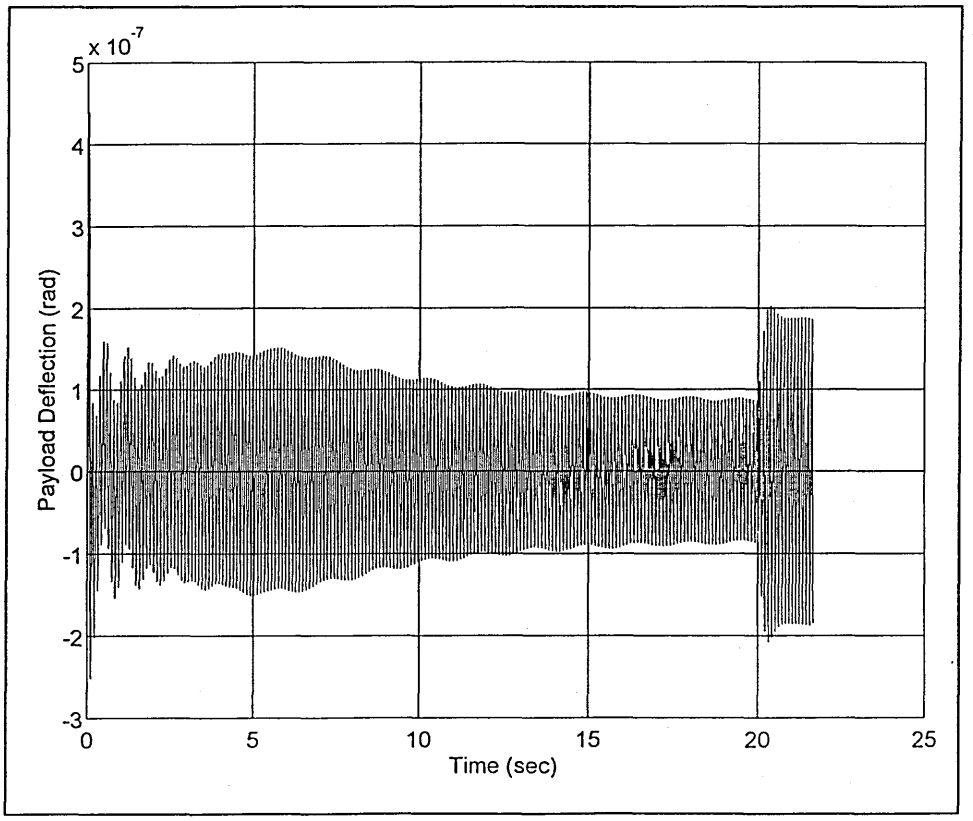

Fig. 6.9 : Controller performance (wheel speed = 60 rad s^{-1}, mc = 0, lr = 0.05)

Figure 6.9 shows that the neural network controller adapted itself effectively on-line to reduce payload disturbances. The network learning algorithm appeared to have converged by the time the controller network was switched off (20 seconds into the simulation). At this point, the controller was reducing payload disturbances by approximately one half. the network had therefore successfully adapted itself to

represent the inverse of the plant dynamics for the particular frequency of disturbance torque input.

Figure 6.10 shows exactly the same simulation as figure 6.9 except that in this case, a learning rate of 0.25 was arbitrarily chosen (compared to the learning rate of 0.05 used in the previous simulation).

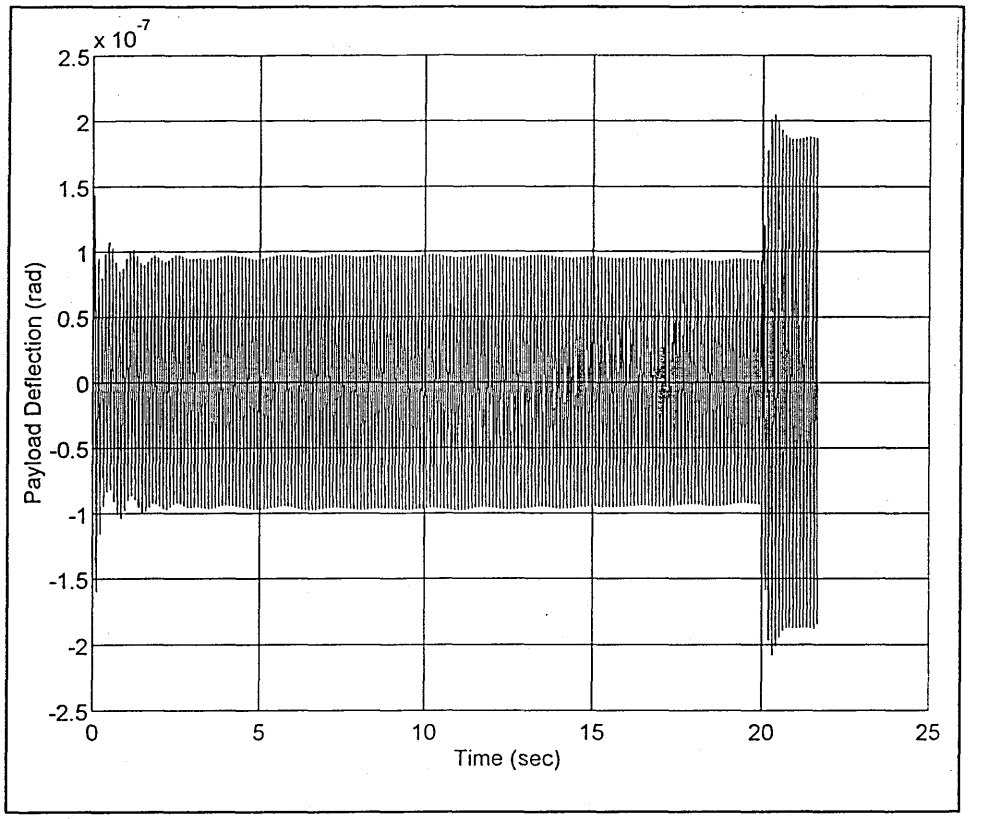

Fig. 6.10 : Controller performance (wheel speed = 60 rad s^{-1}, mc = 0, lr = 0.25)

Once again, the neural network controller quite successfully adapted itself on-line until it represented the inverse dynamics of the system at the frequency of the disturbance torque input. However on this occasion, the neural network learning algorithm converged much more rapidly, indicating that the larger learning rate of 0.25 was much more suitable.

Figure 6.11 shows a repeat of the previous two simulations except in this case a learning rate of 1.3 was arbitrarily chosen.

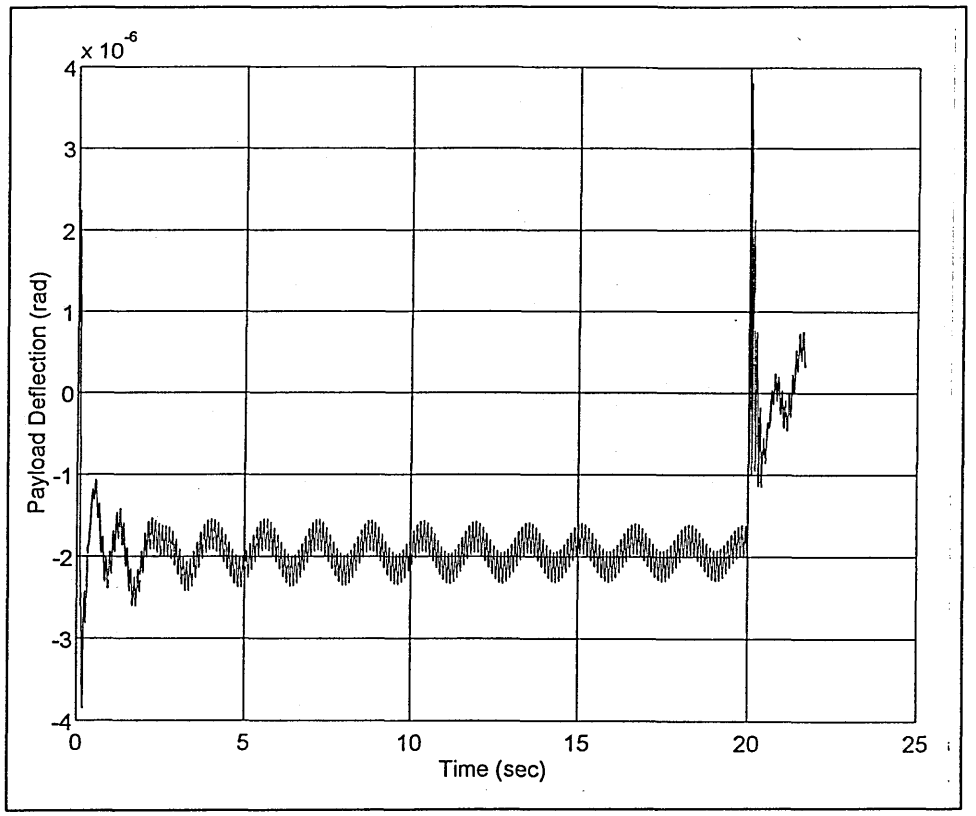

Fig. 6.11 : Controller performance (wheel speed = 60 rad s^{-1}, mc = 0, lr = 1.3)

Unlike in the previous two simulations, the neural network controller in this case performed very poorly. The initial control signals were so poor and the weight changes so large that very large payload deflections initially occurred. The large learning rate ensured that the network overshot its target at each of the simulation time steps when the network weights were updated. The weight changes that occurred as a result during the course of the simulation were very large, and the controller was never really able to recover from its initially poor performance. Finally, when the controller was switched off after 20 seconds, the payload did not get a large control signal to counteract the last large control signal that it received from the neural network controller. The payload was therefore deflected through a large angle, and did not settle before the end of the simulation. This shows how important the values of certain neural network parameters can be, even though there is no way of knowing in advance what the optimum values of these parameters might be.

Figures 6.12 and 6.13 show the response of the plant when the simulations shown in figures 6.9 and 6.10 were repeated using non zero momentum constants in the controller network learning algorithm.

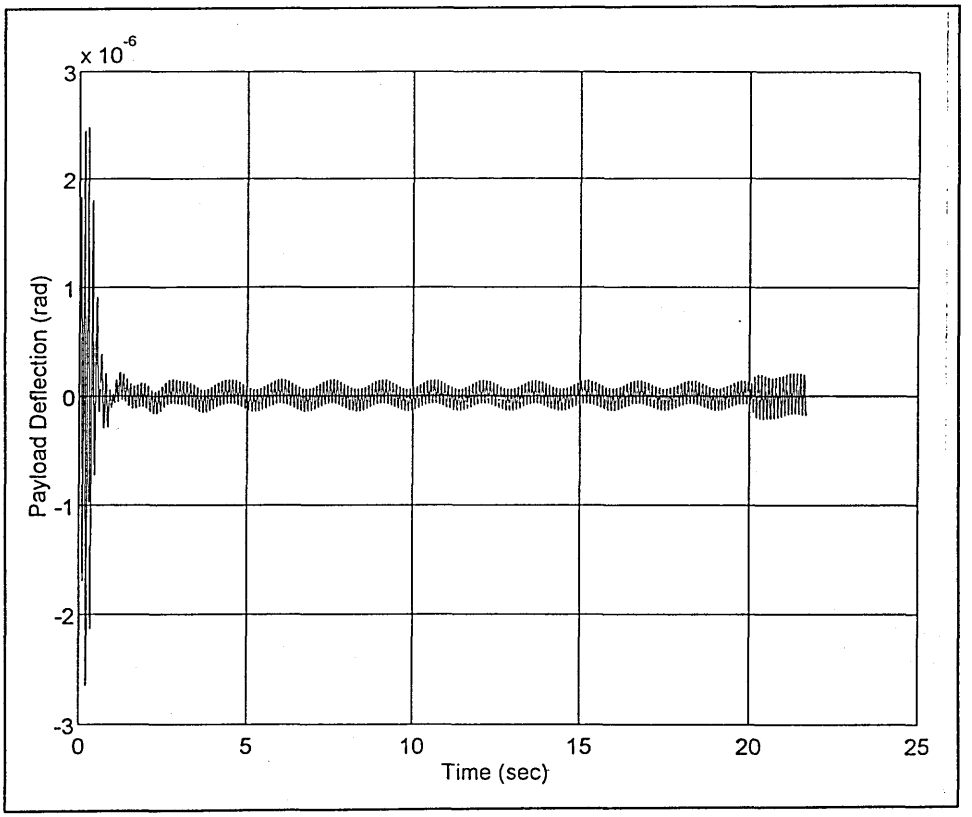

Fig. 6.12 : Controller performance (wheel speed = 60 rad s^{-1}, mc = 0.95, lr = 0.05)

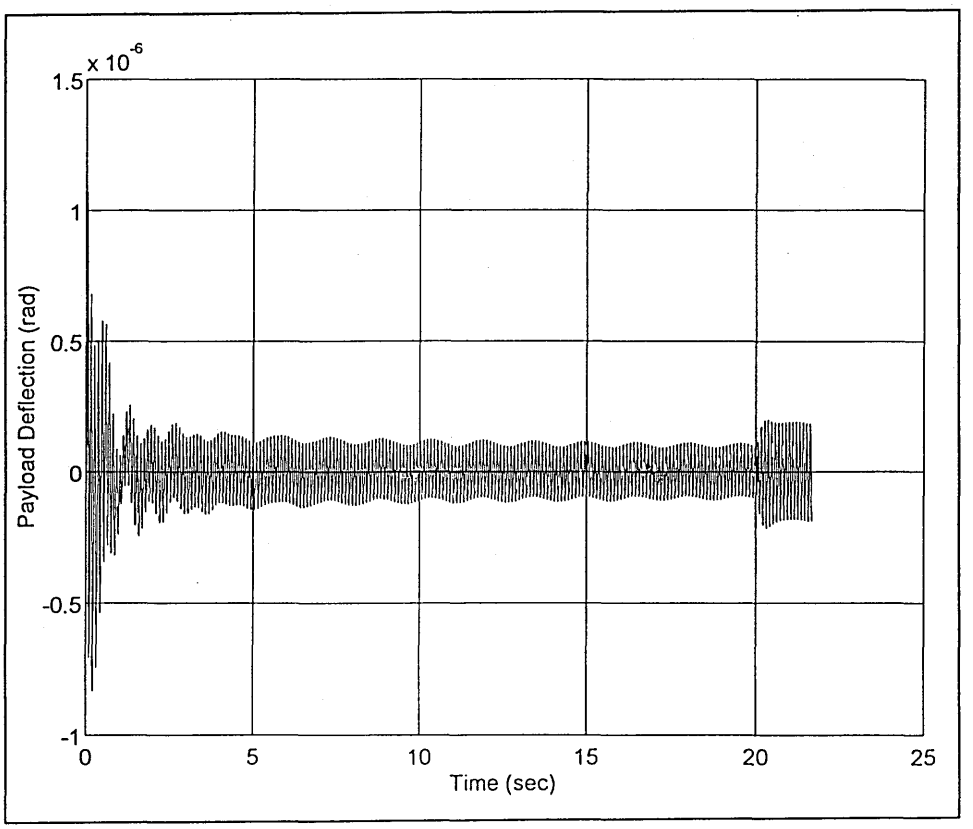

Fig. 6.13 : Controller performance (wheel speed = 60 rad s^{-1}, mc = 0.95, lr = 0.25)

In each of the cases shown in figures 6.12 and 6.13 a momentum constant of 0.95 was chosen, so that network weight changes at each simulation time step were based 95% on previous weight changes and only 5% based on the current network error gradient.

As can be seen from both figures, employing a momentum constant in the controller network learning algorithm resulted in the plant experiencing a severe transient response in each case. These transient responses resulted in such large initial deflections of the payload that low amplitude low frequency payload oscillations were set up. In each case the oscillations did not die out as the simulations continued, even when the controller was eventually switched off after 20 seconds.

The simulations shown in figures 6.12 and 6.13 were repeated using various values for the momentum constant of the controller network. It was found that performance always improved when the value of the momentum constant was reduced, the best results being obtained when a zero momentum constant was used (as shown in figures 6.9 and 6.10).

6.3.2 Controller Performance under Non-Standard Conditions

In the previous section it was shown that the Indirect Model Inversion method was successful in reducing payload disturbances, given a suitable choice of parameters for the neural network learning algorithm. However, all of the simulations that have been discussed so far were performed with a disturbance torque input to the plant that corresponded to a reaction wheel speed of 60 rad s^{-1}. This is the same reaction wheel speed that was used to generate the disturbance torques that were used in training the forward model neural network which plays an integral part in the Indirect Model Inversion control method.

In order to investigate the robustness of the neural network controller that was used for the Indirect Model Inversion control method investigations, a number of additional simulations were carried out using disturbance torque inputs corresponding to various reaction wheel speeds. Figures 6.14 to 6.16 show the response of the payload and the performance of the controller when disturbance torque inputs corresponding to reaction wheel speeds of 50 rad s^{-1}, 40 rad s^{-1}, and 30 rad s^{-1} respectively were used.

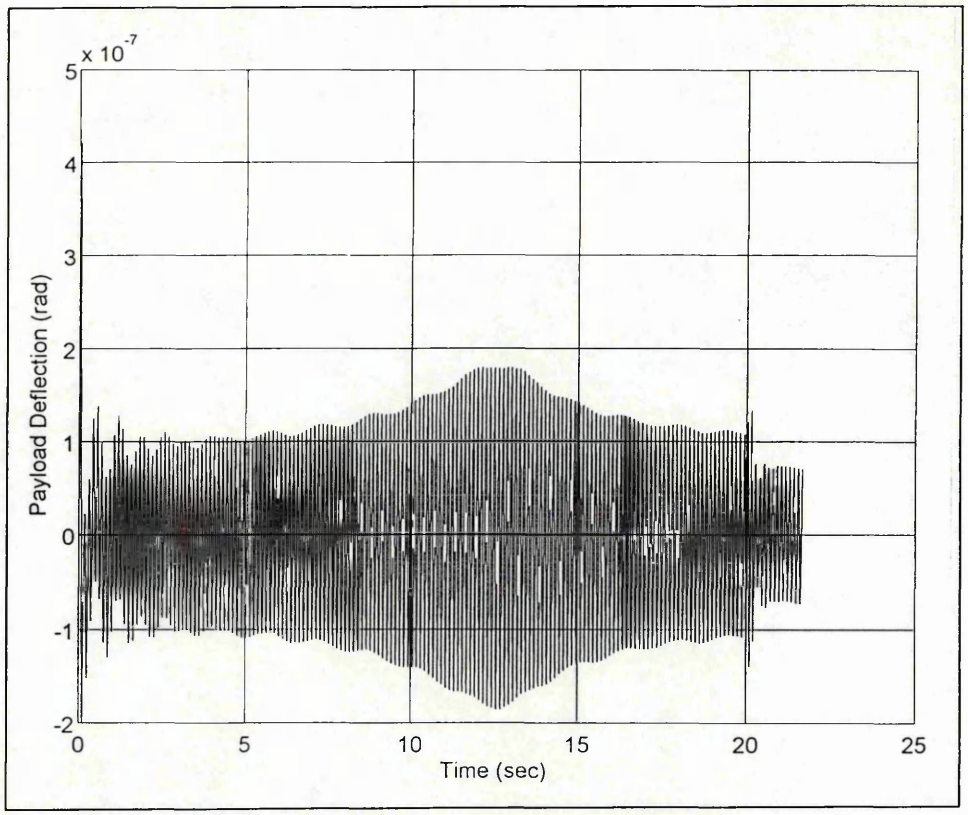

Fig. 6.14 : Controller performance (wheel speed = 50 rad s^{-1}, mc = 0, lr = 0.05)

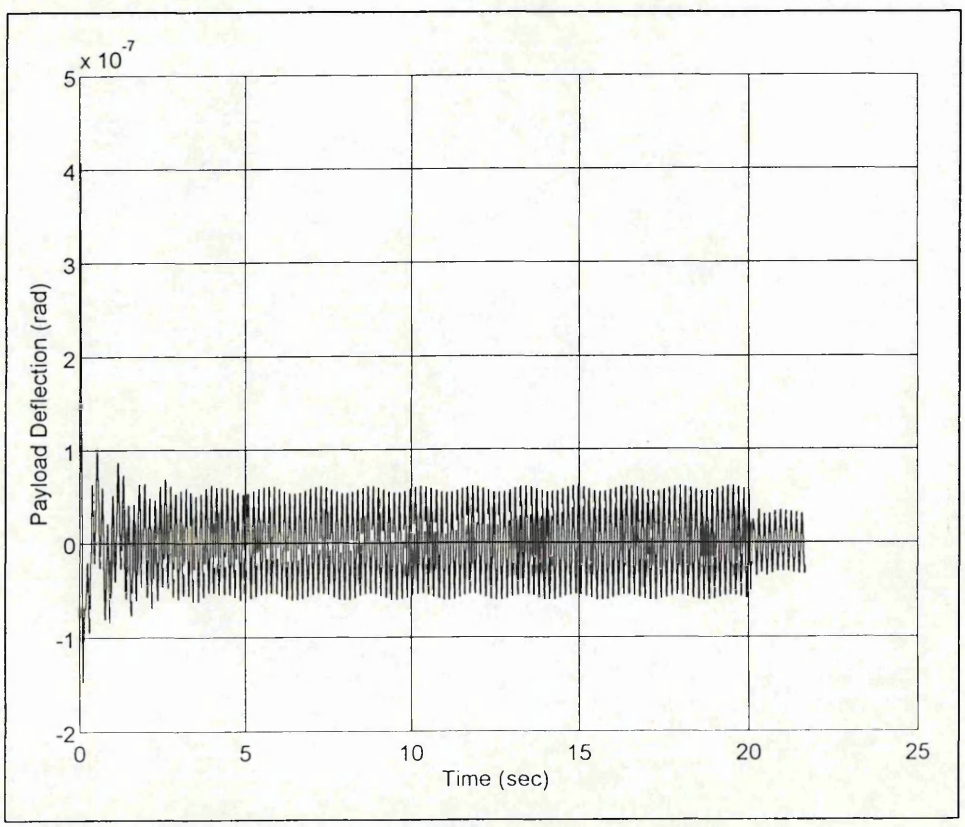

Fig. 6.15 : Controller performance (wheel speed = 40 rad s^{-1}, mc = 0, lr = 0.05)

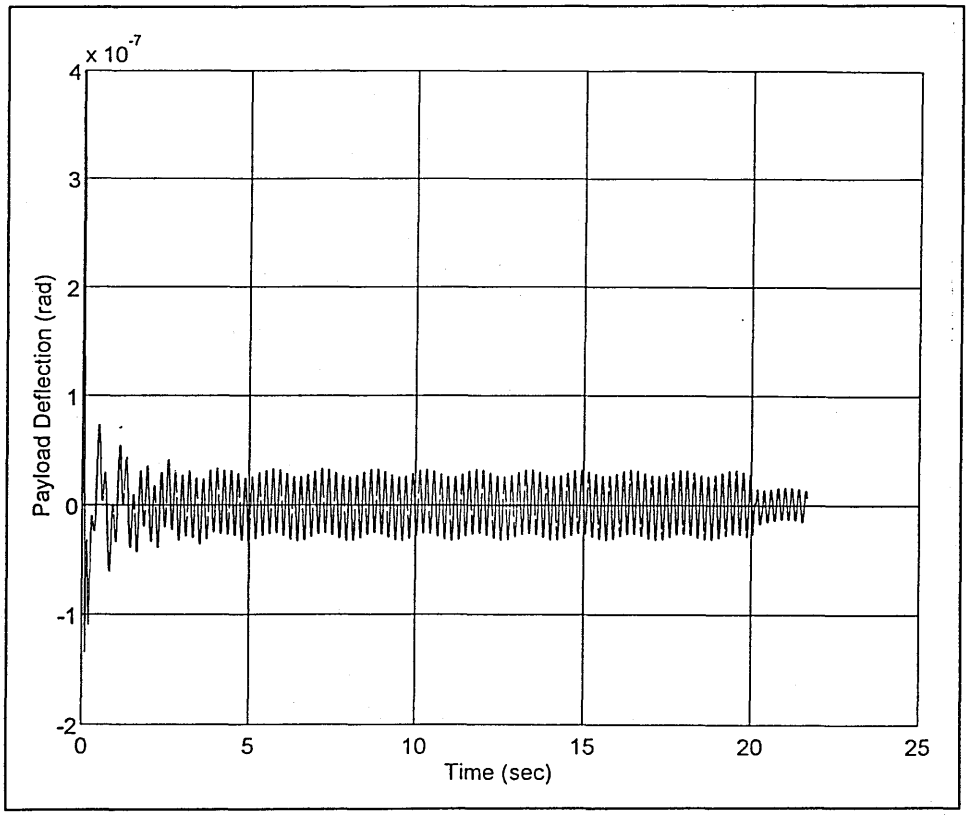

Fig. 6.16 : Controller performance (wheel speed = 30 rad s^{-1}, mc = 0, lr = 0.05)

In each of the three cases shown above, a zero momentum constant and a learning rate of 0.05 were used. Again, the controller was switched on and set to adapt itself on-line for the first 20 seconds of each simulation.

Once again, it can be seen that as the speed of the reaction wheel is lowered, the performance of the controller drops off rapidly. It is not clear whether in the first case shown in figure 6.14 the controller would have converged much further had it been possible to run the simulation for longer. However, in the cases shown in figures 6.15 and 6.16, it is clear that using the neural network controller causes an increase in the deflection of the payload of the spacecraft.

Also of note in each of the simulations shown in figures 6.14 to 6.16 are the low amplitude, low frequency vibrations that are induced in the payload by the neural network controller. Again, this can be put down to the problem of the randomised weights and biases of the network initially causing the controller to provide totally unsuitable control signals that initially generate very large deflections in the payload. As discussed earlier in this chapter, this problem can be overcome to a certain extent by choosing a more appropriate (though ad hoc) value for the learning rate of the network.

In order to further investigate the robustness of the controller, an additional simulation was carried out where the inertia of the spacecraft suddenly changed unexpectedly.

Figure 6.17 shows the performance of an initially untrained controller where the inertia of the spacecraft drops suddenly by 10% after the first 5 seconds. The input torque to the model corresponded to a reaction wheel speed of 60 rad s^{-1}. A zero momentum constant and a learning rate of 0.05 were used.

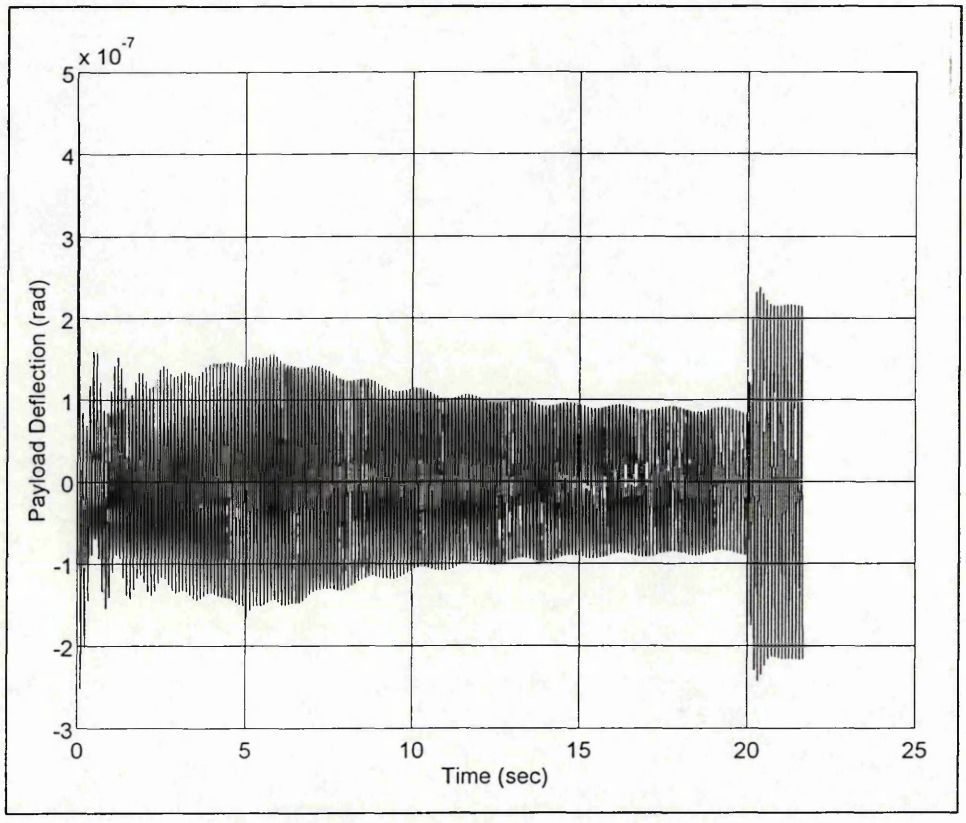

Fig. 6.17 : Controller performance (reduction in spacecraft inertia after 5 seconds)

As can be seen from figure 6.17, making a small change in the inertia of the spacecraft makes little difference to the dynamics of the spacecraft, and the controller is able to perform well, as before.

6.3.3 Controllers with Multiple Neurons in the Input Layer

As part of the investigation into the Indirect Model Inversion control method that was carried out during this work, the effect on controller performance of varying the number of tansigmoidal neurons in the input layer of the controller network was studied.

One of the problems with investigating the performance of controller networks that have large numbers of neurons in their input layers is that the simulations take a considerable amount of time. This is due to the large numbers of weights and biases that must be continually coded and decoded by the S-Function that represents the controller network in Simulink. For this reason, the variation in controller performance with varying numbers of input neurons in the controller network was not extensively studied in this work.

Figures 6.18 to 6.21 show the results that were obtained when controller networks with 3, 5, 7 and 10 tansigmoidal neurons in their input layers were used. In each case a zero momentum constant and a learning rate of 0.05 were used. Although, the controllers were identical in every respect apart from the number of neurons in their hidden layer, it must be remembered that the varying network architectures resulted in varying initial conditions as the initial weights and biases of each of the networks were different.

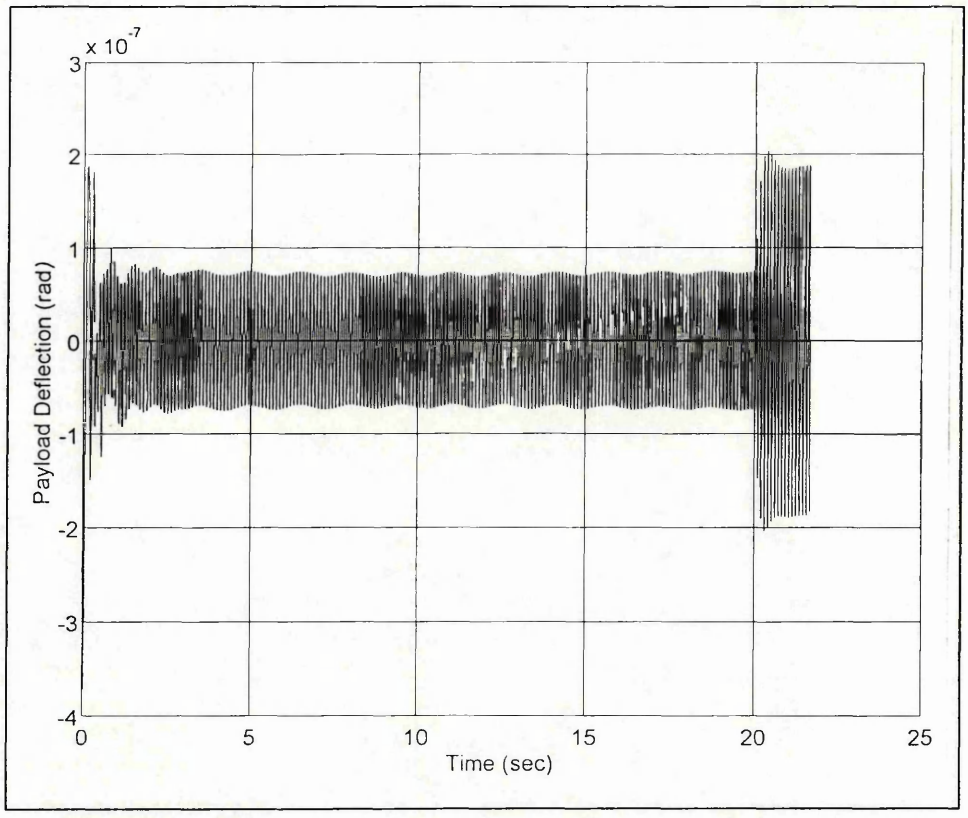

Fig. 6.18 : Controller performance (3 neurons in the hidden layer)

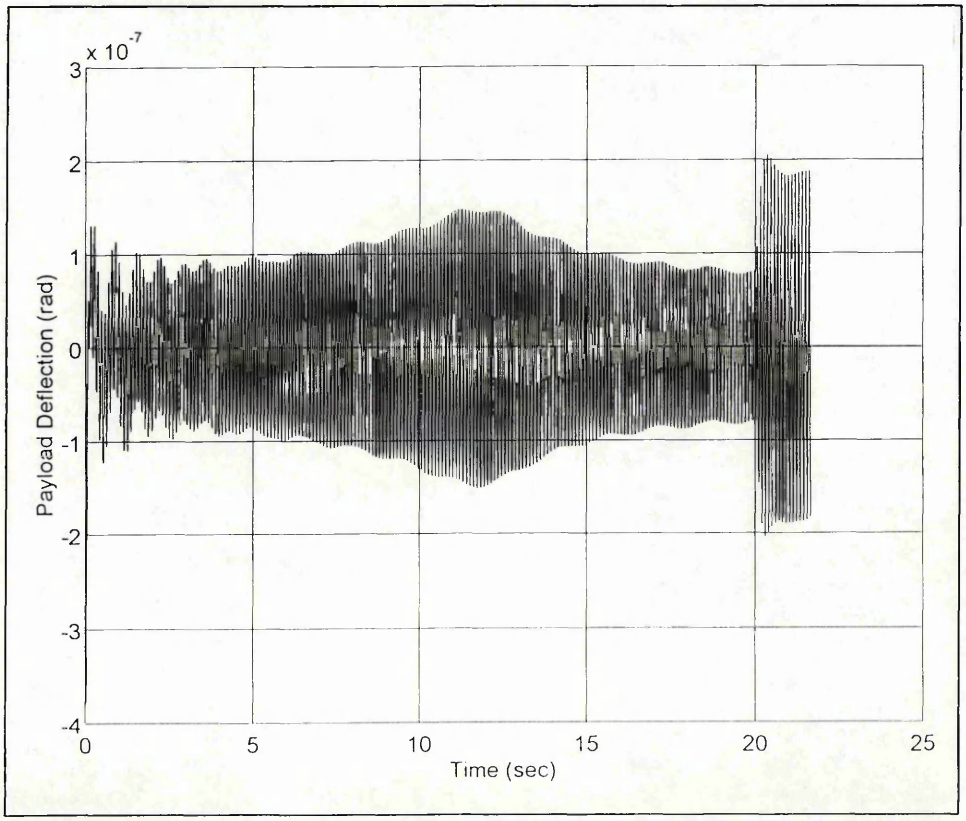

Fig. 6.19 : Controller performance (5 neurons in the hidden layer)

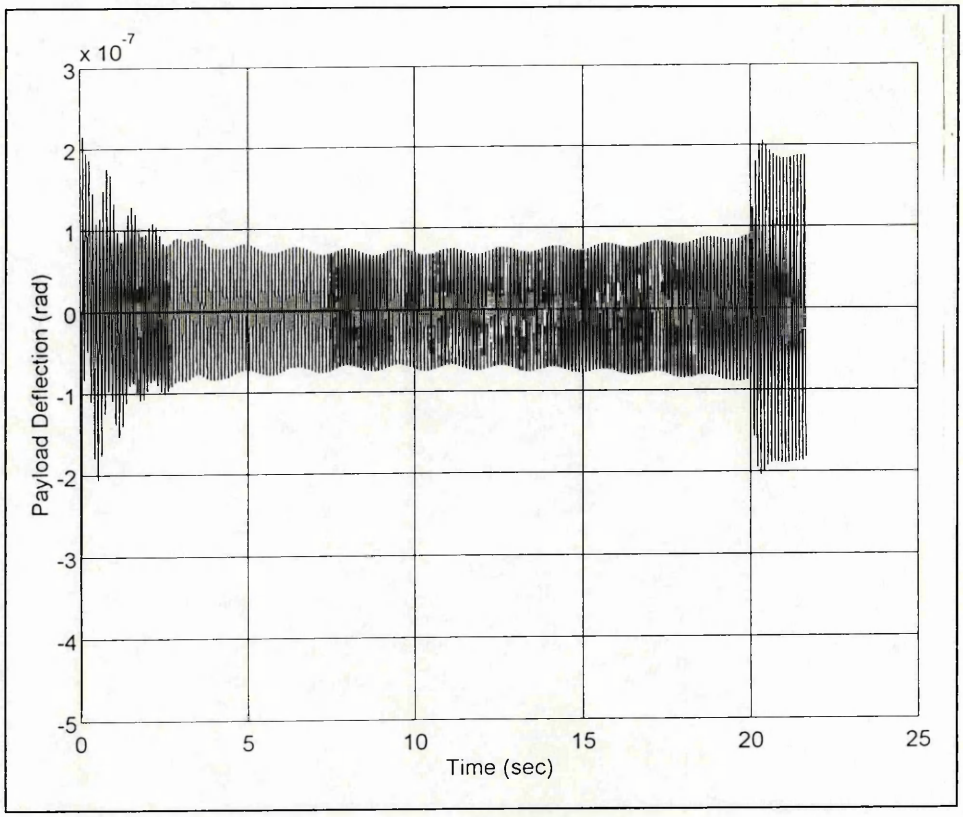

Fig. 6.20 : Controller performance (7 neurons in the hidden layer)

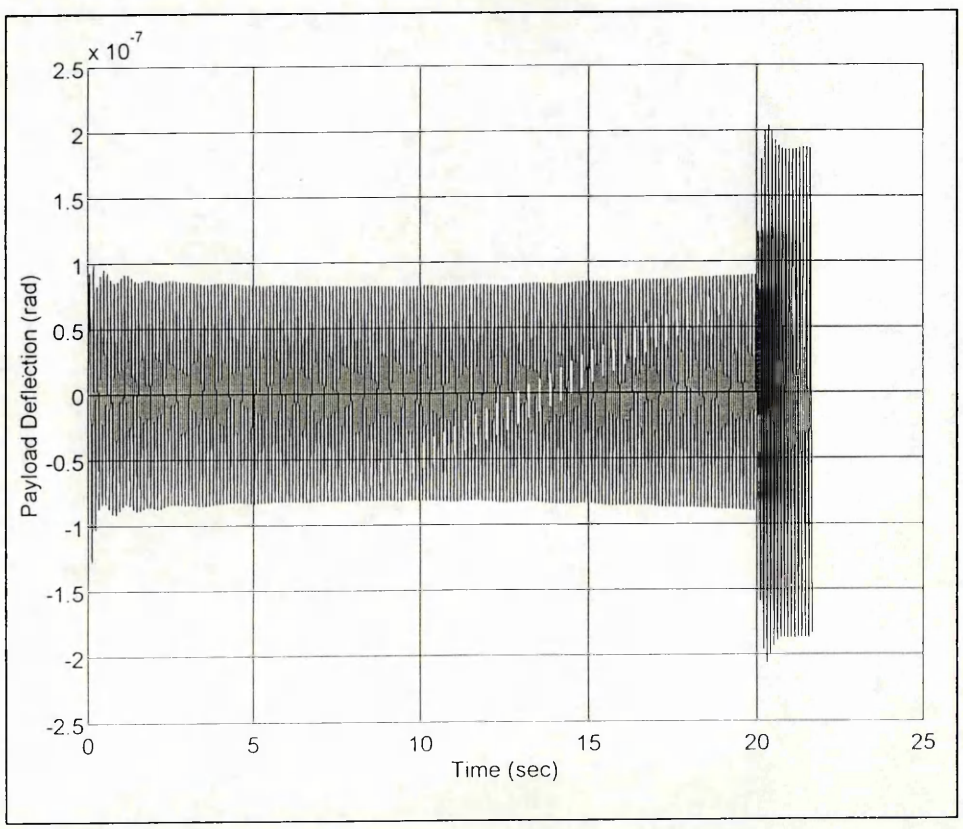

Fig. 6.21 : Controller performance (10 neurons in the hidden layer)

As can be seen from figures 6.18 to 6.21, the number of neurons in the hidden layer of the controller neural network makes little difference to the end result, once the networks in each case have converged. Clearly, there are some differences in the performances of each of the networks, but no general trend is recognisable in increasing the number of hidden neurons in the controller network.

One interesting point to note is that the network with the largest number of neurons (10 neurons) appeared to learn the fastest at the beginning of the simulation. This resulted in a reduced transient response, and thus a smoother performance for the remainder of the simulation. This seems to be in accordance with the generally accepted principle in neural network technology that networks with a large number of neurons generally learn more quickly than networks with fewer neurons.

One possible advantage of using neural networks with more neurons than necessary is that they may be more robust than networks comprising just a single neuron.

In order to investigate the robustness of a neural network employing more than one neuron, a further simulation was carried out where one of the neurons of an initially trained controller was forced to fail after 5 seconds. This was performed by editing the S-Function code so that the weights and bias of the neuron were set to zero for $t > 5$ seconds. Figure 6.22 shows the performance of this neural network controller.

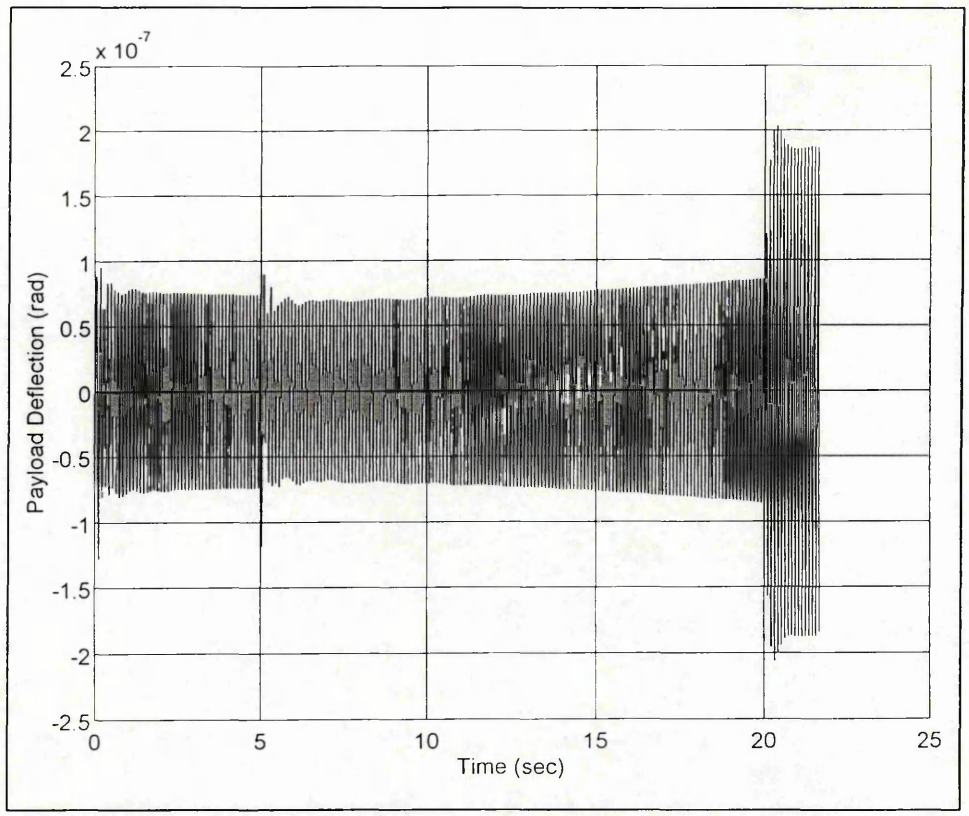

Fig. 6.22 : Controller performance (5-neuron network, one neuron fails)

As can be seen from figure 6.22, a single neuron failure causes a glitch in the payload deflection, but the neural network quickly re-adapts itself to provide suitable control signals. this clearly shows that there is an advantage in using self-adapting neural networks with more neurons than strictly necessary. The robustness of the neural network controller to individual component failure has clear advantages for space-based control applications.

6.3.4 Technical Note Concerning Software Failures

One of the main problems that was encountered in the work carried out in this thesis concerned the consistent failure of the Matlab software when long simulations were carried out.

Often, it would have been useful to have been able to carry out simulations for longer than 22 seconds to see if a controller would eventually converge. However, this was not possible as Matlab always crashed, returning an "Out of Memory Error". This problem

also occurred if a series of short simulations were carried out in succession. It was therefore necessary to quit the Matlab application at the end of each simulation before a new one was started. These problems clearly should not have been encountered as the machine used for the work in this thesis was a 486DX-33 with 8mb of memory.

To try to overcome these problems, the amount of "Virtual Memory" allocated to the Windows 3.1 operating system was increased. Although this did have some effect initially, it was found that once a particular limit (of about 19mb) had been exceeded no further gains could be obtained.

By allocating large amounts of virtual memory to the Windows operating system, another problem was introduced which was that during the simulations, Matlab spent a considerable amount of time writing to the hard disk. This drastically increased the time it took to run the simulations.

Clearly, the problems experienced here should not have been encountered. The amount of data that needed to be stored during the simulations was far less than the memory that should have been available to Matlab. Nobody else at Cranfield seemed to have any experience of this problem. However, towards the end of this work, a personal communication with a contact in industry revealed that similar problems have been encountered elsewhere, and that the problem does not occur with the next Matlab upgrade. It is therefore believed that the problem arises from a bug in the way that Matlab v4 manages memory.

7 Conclusions

The work that was carried out in this thesis was successful in a number of respects. Firstly a spacecraft model and a corresponding spacecraft control problem were successfully represented in the Simulink environment. Secondly, two neural network control methods known frequently as Direct Model Inversion and Indirect Model Inversion were successfully applied to the spacecraft control problem. Furthermore, the neural network controllers that were produced showed robust behaviour under certain conditions.

7.1 Direct Model Inversion

The Direct Model Inversion method was successfully used to train a neural network to model the inverse dynamics of the plant to be controlled. When the inverse model was placed in the feedback path of the plant, payload disturbances could be reduced by a factor of one half, in accordance with predictions.

Despite the apparently successful results, the neural network inverse model only closely represented the inverse dynamics of the plant when a sinusoidal disturbance torque input of a certain frequency was applied to the plant. This particular frequency was the same as that used to generate the training vectors that were used to train the neural network controller off-line. When the frequency of the disturbance torque that was applied to the plant was reduced, the performance of the neural network controller deteriorated rapidly.

7.2 Indirect Model Inversion

The Indirect Model Inversion method was also successfully used to produce a neural network that could be used to control the spacecraft plant. Again an inverse model of the plant to be controlled was produced that could reduce payload deflections by one half when placed in the feedback path of the plant.

Once again, these apparently successful results were disappointing. The inverse model neural network only closely resembled the actual plant inverse at a particular frequency. This frequency corresponded to the frequency that was used to generate the training vectors that were used to train the forward model neural network. When the frequency of the disturbance torque input to the plant was reduced, the performance of the neural network controller deteriorated rapidly.

Another cause for disappointment was that it was originally envisaged that something more than a simple inverse plant model could be obtained using the Indirect Model Inversion method. It was hoped that the controller network would adapt itself on-line in such a way that it would be able to provide optimum control signals so that payload deflections would be quite considerably reduced. However, this was not to be the case.

It is possible that the disappointing results obtained using Indirect Model Inversion arose from the way in which the weights and biases of the neural network controller were updated on-line. At any instant in time, the neural network controller was adapted in such a way that the single plant error at that instant in time was minimised. However, the updated set of weights and biases may not have been the most appropriate to provide a control signal at the next time step of the simulation. The controller network was therefore continually being updated in response to a local error (in time) rather than a global error. A good way of getting around this problem would have been to store some record of say the last 100 plant errors, and then to update the network in a way that minimises all of these errors rather than just the most recent one. Network weight and bias updates could then be set to occur every 100 simulation time steps, for example. This form of on-line batch learning may have been able to produce a more effective controller that could have reduced payload deflections more effectively.

Some of the successes achieved with the Indirect Model Inversion method are concerned with the results that were obtained using controller networks with multiple neurons in their input layers. The following results are of note:

(i) Self-adapting neural network controllers with at least several neurons in their input layer were not particularly affected by the in-operation failure of one of their neurons.

(ii) The neural network with the largest number of neurons was the quickest to learn at the beginning of the simulation, resulting in an improved transient response, and smoother overall performance.

The first of these points has particular relevance to space-based applications, where redundancy and robust performance are of paramount importance.

The second point seems to be in accordance with the generally accepted principle in the field of neural network technology that larger networks require less training.

7.3 General Conclusions

Although some relatively successful results were obtained during the course of this work, a number of conclusions can be drawn that raise concerns about the application of neural network technology in this context. These are :

(i) In all of the simulations carried out in this work, a better control performance could be obtained by using a simple positional feedback gain term. (For this reason, it was felt that in this thesis, it would not be worthwhile comparing the performance of the neural network controllers with the performance of a controller of conventional design).

(ii) The neural network controllers that were produced were not effective over a wide enough range of disturbance torque frequencies to justify their use as controllers.

(iii) The whole of neural network technology is based on a trial and error methodology that is completely at odds with the thorough engineering principles that are universally adopted in the space industry.

Although these conclusions are quite negative, it should be pointed out that they apply to the work carried out in this thesis and to the current state of neural network technology. These conclusions do not necessarily suggest that there will be no place for neural network control techniques in the space missions of the future. However, the technology is not yet sufficiently developed for any long term conclusions to be drawn.

7.4 Suggestions for Further Work

Now that the work carried out in this thesis has been completed, an number of suggestions for further work are immediately apparent.

Probably the most significant area in which further investigations would be most worthwhile concerns the Indirect Model Inversion control method and its related problems (as discussed in section 7.2 above). The controller network could be further developed so that error vectors could be batched on-line for the purpose of batch learning. This could be done by carefully modifying the S-Function program that was used to carry out the on-line functions of the controller network. By doing this, the controller would adapt itself in such a way that a number of plant errors would be minimised rather than just the current plant error. This might result in the controller learning on-line to provide more suitable control signals that might in turn produce greater reductions in payload deflections.

Another area in which further study might prove fruitful is the area of Radial Basis Function networks. As discussed in chapter 2, Radial Basis Function networks are networks of neurons that make use of symmetrical (often Gaussian) transfer functions. Like backpropagation networks using tansigmoidal neurons, Radial Basis Function networks can be used as universal function approximators, but have the added feature of possessing a guaranteed global minimum. Radial Basis Function networks were not investigated in this work as they are not supported in the Neural Network Toolbox that is available with Matlab v4. However, it is understood that neural Network Toolbox that is provided with the subsequent Matlab upgrade does support the use of Radial Basis Function networks. It is also understood that networks of this kind can be simulated considerably faster than the backpropagation networks studied here.

Further improvements that can be suggested here include developing the spacecraft model further. For example, a reaction wheel model could be developed and incorporated into the spacecraft model used here. Other sensor models could also be developed and incorporated into the spacecraft model. These could comprise traditional positional sensors or perhaps rate gyros. Also, the inner control loop presented here could be studied in the wider context of the outer control loop that would be used in conjunction with a traditional (e.g. PID) control law for coarse attitude control.

Finally a further useful extension to the work presented here would be to incorporate some non-linearities into the spacecraft plant model. As conventional control algorithms have difficulties in controlling non-linear plants, a non-linear spacecraft plant model would provide a better testing ground for neural network control methods.

Appendix A: Matlab Programs

In this appendix, a listing of the two Matlab language programs that were used in the Indirect Model Inversion control simulations are given.

The first program was used to initialize the controller network, and could be run directly from the Matlab workspace or by clicking on an icon in Simulink.

The second program was used by the S-Function that represented the controller network in the Simulink environment . At each time step of a simulation, the program first decodes its inputs (which are the weights and biases of the network and the parameters used by the network's learning algorithm). The program then calculates the neural network output and output error, and updates network weights and biases according to the values of the parameters of the learning algorithm. Finally, the program re-codes the weights, biases, network output and output error and passes them back to Simulink.

(i) Program to initialise the architecture of the neural network controller.

```
% Program to initialize the ANN architecture

fprintf('\n');
clc
fprintf('ANN architecture setup program\n');
fprintf('==============================\n');
inp=input('    How many elements are there in the ANN input vector ? ');
hid=input('    How many tansigmoidal neurons do you require in the hidden layer ? ');
delay=input('    Enter delay required between sampled plant inputs (sec) : ');

answer=input('    Do you wish to randomize initial weights and biases (y/n) ? ','s');
if answer=='Y'
        answer='y';
end

% Calculate total no. of weights and biases for Mux & Demux blocks
weightsize=(inp*hid)+(2*hid)+1;

clc
fprintf('The following ANN architecture has been configured :');
fprintf('\n=====================================================');
fprintf('\n\n    No. of tansigmoidal neurons in the input layer : %g',hid);
fprintf('\n    No. of inputs to the input layer : %g',inp);
fprintf('\n    No. of linear neurons in the output layer : 1');
```

```
if answer=='y'
        weights=rands(1,(inp*hid)+(2*hid)+1);
        fprintf('\n\n    Weights and biases have been randomized in the region +/- 1');

else
        fprintf('\n\n    N.B. Weights and biases have not been initialized\n     - make
sure that they are available in the workspace\n\n');
end
clear answer
fprintf('\n\n    Press enter to return to Simulink');
```

(ii) Program called by the S-Function that represented the neural network controller in Simulink.

```
function [sys,x0]=sfun_ann(t,x,u,flag,weights,weightsize,inp,hid,delay);
% m-file called by ANN S-function
% The following variables are passed to the S-function from the workspace
% weights    : initial weights from workspace
% weightsize : Total no. of weights and biases
% hid        : No. of neurons in the hidden layer
% inp        : No. of inputs to the hidden layer
% delay      : Simulation step size

if flag==3
        % Calculate end markers for encoded inputs
        W1end=hid*inp;
        B1end=W1end+hid;
        W2end=B1end+hid;
        B2end=W2end+1;
        Pend=B2end+inp;

        % De-code weights & biases
        if t==0
                % Get weights from workspace and output them
                sys=[weights,0,0];
        else
                % load weights & biases from S-function input vector
                W1=ones(hid,inp);
                for row=1:hid
                        for col=1:inp
                                W1(row,col)=u(((row-1)*inp)+col,1);
                        end
                end
                B1=u(W1end+1:B1end,1);
                W2=u(B1end+1:W2end,1)';
                B2=u(B2end,1);

                % De-code remaining inputs
                P=u(B2end+1:Pend,1);
                T=u(Pend+1,1);
                mom=u(Pend+2,1);
                lr=u(Pend+3,1);
```

```
                eratio=u(Pend+4,1);
                lflag=u(Pend+5,1);

                % Calculate ANN output
                A1=tansig(W1*P,B1);
                A2=purelin(W2*A1,B2);
                E=T-A2;
                D2=E;
                D1=deltatan(A1,D2,W2);

                if lflag==0
                        % Don't calculate new weights
                        wout=u(1:B2end,1);
                        sys=[wout',A2,E];
                else
                        % Calculate new weights
                        dW1=0*W1;
                        dB1=0*B1;
                        dW2=0*W2;
                        dB2=0*B2;
                        [dW1,dB1]=learnbpm(P,D1,lr,mom,dW1,dB1);
                        [dW2,dB2]=learnbpm(A1,D2,lr,mom,dW2,dB2);
                        W1new=W1+dW1;
                        W2new=W2+dW2;
                        B1new=B1+dB1;
                        B2new=B2+dB2;

                        % Check new error, disregard new weights if too large
                        A2new=purelin(W2new*tansig(W1new*P,B1new),B2new);
                        Enew=T-A2new;
                        if (abs(Enew/E))<eratio==1
                                W1=W1new;
                                B1=B1new;
                                W2=W2new;
                                B2=B2new;
                        end

                        % En-code output vectors
                        W1=W1';
                        W1=W1(:);
                        W1=W1';
                        B1=B1';
                        sys=[W1,B1,W2,B2,A2,E];
                end
        end
elseif flag==4
        ts=delay;
        ns=t/ts;
        sys=(1+floor(ns+1e-13*(1+ns)))*ts;
elseif flag==0
        nin=weightsize+inp+5;
        nout=weightsize+2;
        sys=[0,0,nout,nin,0,0];
else
        sys=[];
end
```

References

[1] McCulloch, W.S., and Pitts, W., 1943, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, **5**, 115-133.

[2] Cybenko, G., 1989, "Approximation by superposition of a sigmoidal function", *Mathematics of Control, Signals and Systems*, **2**, 303-314.

[3] Funahashi, K., 1989, "On the approximate realisation of continuous mappings by neural networks." *Neural Networks*, **2**, 183-192.

[4] Hornik, K., Stinchcombe, M., and White, H., 1989, "Multilayer feedforward networks are universal function approximators." *Neural Networks*, **2**, 359-366.

[5] Stinchcombe, M., and White, H., 1989, "Universal approximation using feedforward networks with non-sigmoid hidden layer activation functions." *Proc. Intl. Joint Conf. on Neural Networks*, **1**, 613-617.

[6] Game, G.W., 1992, "The application of neural networks to vibration suppression in spacecraft structures." *Proc. of the 2nd Int. Conf. on Dynamics and Control of Structures in Space*, Cranfield University, Sept. 1993.

[7] The Mathworks Inc., 1993, "Neural Network Toolbox User's Guide."

[8] Chen, S., Billings, S.A., Cowan, C.F.N., and Grant, P.M., 1990, "Non-linear systems identification using radial basis functions." Int. J. Sys. Sci. **21**, 2513-2539.

[9] Leonard, J.A. and Kramer, M.A., 1991, "Radial basis function networks for classifying process fault." *IEEE Control Systems*, **3**.

[10] Park, J., and Sandberg, I.W., 1991, "Universal function approximation using radial basis function networks." *Neural Computation*, **3**, 246-257

[11] Ahmed-Zaid, F., Ioannou, P., Polycarpou, M.M., and Youssef, H.M., 1992, "Identification and control using radial basis function networks." *Proc. AIAA Guid., Nav., and Control Conf.*

[12] Youssef, H.M., and Juang, J., 1993, "Multiple radial basis function networks in modelling and control." *Proc. AIAA Guid., Nav., and Control Conf*, 285-289

[13] Rosenblaat, F., "The Perceptron: A probablistic model for information storage and organisation in the brain." *Psychological Review*, **65**, 386-408

[14] Rosenblaat, F., 1962, "Principles of neurodynamics: Perceptrons and the theory of brain mechanisms." *Washington, D.C.: Spartan Books.*

[15] Minsky, M.L., and Papert, S., 1969, "Perceptrons: An introduction to computational geometry." *MIT Press, Cambridge, MA.*

[16] Widrow, B., and Hoff, M.E., 1960, "Adaptive switching circuits." *1960 IRE WESTCON Conv. Record*, **4**, 96-104.

[17] Widrow, B., and Stearns, S.D., 1985, "Adaptive signal processing." *Prentice-Hall, Engelwood Cliffs, N.J.*

[18] Widrow, B., 1986, "Adaptive inverse control." *Adaptive Systems in Control and Signal Processing, International Federation of Automatic Control, July 1986.*

[19] Nguyen, D., and Widrow, B., 1989, "The truck backer-upper: An example of self-learning in neural networks." *Proc. Int. Joint Conf. on Neural Networks, Vol II, 357-363.*

[20] Nguyen, D., and Widrow, B., 1990, "Neural networks for self-learning control systems." *IEEE Control Systems Magazine*, April 1990, 18-23.

[21] Rumelhart, D.E., Hinton, G.E., and Williams, R.J., 1986, "Learning internal representations by error propagation." *Parallel Distributed Processing : Explorations in the Microstructures of Cognition, Vol. 1: Foundations*, 318-362.

[22] Rumelhart, D.E., Hinton, G.E., and Williams, R.J., 1986, "Learning internal representations by the back-propagation of error." *Nature*, **323**, 533-536.

[23] Kohonen, T., 1984, "Self organisation and associative memory." *Springer Verlag, Berlin.*

[24] Kohonen, T., 1988, "An introduction to neural computing." *Neural Networks, Vol. 1, No. 1.*

[25] Hopfield, J.J., 1982, "Neural networks and physical systems with emergent collective computational abilities." *Proc. Nat. Acad. Sci.*, **79**, 2554-2558.

[26] Hopfield, J.J., Feinstein, D.I., and Palmer, R.G., 1983, "Unlearning has a stabilizing effect in collective memories." *Nature*, **304**, 158-159.

[27] Hopfield, J.J., 1984, "Neurons with graded response have collective computational properties like those of two-state neurons." *Proc. Nat. Acad. Sci.*, **81**, 3088-3092.

[28] Hopfield, J.J., and Tank, D.W., 1985, "Neural computation of decisions in optimization problems." *Biological Cybernetics*, **52**, 141-152.

[29] Narenda, K.S., and Parthasarathy, K., 1990, "Identification and control of dynamical systems using neural networks." *IEEE Transactions on Neural Networks* Vol. 1, **1**, 4-26.

[30] Ichikawa, Y., and Sawa, T., 1992, "Neural network application for direct feedback controllers." *IEEE Transactions on Neural Networks* Vol. 3, **2**, 224-231.

[31] Levin, A.U., nad Narendra, K.S., 1993, "Control of nonlinear dynamical systems using neural networks: Controllability and stabilization." *IEEE Transactions on Neural Networks* Vol. 4, **2**, 192-205.

[32] Venugopal, K.P., and Smith, S.M., 1993, "Improving the dynamics response of neural network controllers using velocity reference feedback." *IEEE Transactions on Neural Networks* Vol. 4, **2**, 355-357.

[33] Guez, A., Elibert, J.L., and Kam, M., 1988, "Neural network architecture for control." *IEEE Control Systems Magazine*, April 1988, 22-25.

[34] Psaltis, D., Sideris, A., and Yamamura, A.A., 1988, "A multilayered neural network controller." *IEEE Control Systems Magazine*, April 1988, 17-20.