NEUROMORPHIC REGULATION OF DYNAMIC SYSTEMS USING BACK PROPAGATION NETWORKS

by

ROBERT M. SANNER

Bachelor of Science in Aeronautics and Astronautics
Massachusetts Institute of Technology
(1985)

Submitted in Partial Fulfillment of
the Requirements for the Degree of

MASTER OF SCIENCE IN
AERONAUTICS AND ASTRONAUTICS

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May, 1988

Signature of Author _____
Department of Aeronautics and Astronautics
May 15, 1988

Certified by _____
Professor David L. Akin
Thesis Supervisor
Department of Aeronautics and Astronautics

Accepted by _____
Professor Harold Y. Wachman
Chairman, Departmental Graduate Commitee
Department of Aeronautics and Astronautics

NEUROMORPHIC REGULATION OF DYNAMIC SYSTEMS USING BACK PROPAGATION NETWORKS

by

Robert M. Sanner

Submitted to the Department of Aeronautics and Astronautics
on May 11, 1988 in partial fulfillment of the requirements for
the degree of Master of Science in Aeronautics and Astronautics

Abstract

A series of experiments has been completed in which the ability of modified back propagation neural networks to learn to regulate dynamic systems was systematically evaluated. This research has led to the development of a new type of learning controller, known as the *neuromorphic controller* (NMC). The NMC algorithm uses this modified back propagation methodology to teach neural networks to construct mappings from the current state of the plant to the control actions required in order to maintain the output of the plant at a specified value. For this algorithm it is assumed that neither the network nor the teacher have any *a priori* knowledge of the dynamics of the plant to be controlled. Thus, unlike classical back propagation would require, the NMC is not explicitly shown a control law to emulate, but rather forms its own control law based upon criticism of its behavior by the teacher. The control laws developed by the NMC, and hence the closed loop response of the dynamic system, can be shaped by adjusting the parameters with which the teacher computes its criticism.

This algorithm has been simulated in software and tested on several second and third order, linear and nonlinear dynamic systems using both linear and bang-bang actuation. It has been observed that the control laws constructed by the NMC arise through the tuning of the synaptic weights in response to the correlation of criticism issued by the teacher with the evolution of the plant states during the network's training phases. Through this synaptic tuning mechanism, the individual neurons become sensitized to different states of the plant, effectively becoming adaptive feature detectors on the state space; this controls how and when each neuron contributes to the control law. Often this process results in control laws which are linear in a wide region of the state space; many times, however, the network implements a nonlinear control logic which is quite effective in, for example, suppressing noise and overcoming and exploiting plant and actuator nonlinearities. It is further demonstrated that the resulting controller is robust to damage in the network elements, and can discern which of a set of exogenous stimuli is relevant to solving the control problem.

Thesis Supervisor: Professor David L. Akin
Title: Rockwell Assistant Professor of Aeronautics and Astronautics

Acknowledgements

This thesis is dedicated to my parents, whose love and encouragement over the past years have given me the strength to keep chasing my dreams, wherever they may lead me. I can only hope I give back to them some small fraction of what they have given to me.

Throughout my three (so far!) years as a graduate student here at M.I.T., I have had the great fortune to be associated with Professors who have not only tolerated, but also encouraged my nutty ideas. To Prof. Markey (advisor to ClasCon: the thesis that almost was), I will be always indebted for allowing me the opportunity to design software for his course, and for his consistently excellent advice and friendship. Prof. Akin, who was willing to take a chance with a newcomer in his laboratory, and allowed me to chase after an (at the time) totally obscure topic, can never be thanked enough. I only hope my next few years with him will be as exciting as this first has been.

Thanks are especially due to my good friend Stuart (Sport Death) Brorson, who first introduced me to neural networks, and in whose kitchen I first had the idea for the research described in this thesis. And speaking of Stuart, I must offer my most profound apologies to the fine people of Cambridge, whom Stuart and I no doubt annoyed during our loud, protracted arguments on this topic at the S&S and CC. Thanks also to my friend and roommate Jeff Inman, whose advice on an object oriented implementation of the network equations was extremely helpful in getting this research started. If only *I'd* had a Lisp Machine to hack on!! (are you listening, Dave?)

And last, but never least, I have to thank everyone who hangs his or her hat in 33-407. You have all helped me more than you know: some with tangible, technical matters, others by just making me laugh when I most needed it.

This research was made possible by the NASA Office of Aeronautics and Space Technology, under NASA Grant NAGW-21.

Table of Contents

Table of Figures

Chapter 2:

Chapter 3:

Chapter 4:

Chapter 1: Introduction

Adaptation. This quality, seemingly so effortless for biological systems, is maddeningly difficult to instill in man-made machines. The two most promising approaches, adaptive control theory and the theories of artificial intelligence, have experienced serious difficulties implementing even small examples of useful learning automata. Adaptive control theories can be proven stable for only a very small class of possible dynamic systems , and even these can, under unfavorable circumstances, become unstable (Rohrs *et al.*, 1982). Artificial intelligence approaches have had great laboratory success, but usually with computer environments so restrictive that they have limited practical utility. Reliable learning automata thus remain a distant dream.

And yet machines which can learn from and adapt to their environments will be crucial in helping mankind explore and exploit space. In 1982 the MIT Space Systems Laboratory (SSL) conducted the ARAMIS (Automation, Robotics, And Machine Intelligence Systems) study for NASA. This research concluded that a proper mixture of machine intelligences and automata to augment human capabilities would significantly reduce the cost and increase the productivity of certain space activities (Miller *et al.*, 1983). Two areas targeted particularly for future research by this study were teleoperation and expert systems: teleoperation in order to better understand how to choreograph activities between semi-autonomous machines and their remote human operators, and expert systems because, in the words of the study, "as spacecraft complexity increases the prediction of all [possible] failure modes and effects becomes combinatorially enormous...the expert system may be the best method to deal with spacecraft failures" (Miller *et al.*, 1983). Machines which can recognize and respond to unforeseen circumstances (fault tolerance) and adapt to new tasks and operating conditions will be necessary adjuncts to any human space presence.

Recent research conducted by the SSL bears out these conclusions of the ARAMIS report. Even limited amounts of machine autonomy yield large performance increases in the tasks of structural assembly and satellite docking and retrieval conducted in a simulated space environment (Anderson, 1988; Tarrant, 1987). Expert systems have also been developed to assist with these and related tasks (Viggh, 1988; Kurtzman, 1987). However, even these limited amounts of machine intelligence and autonomy require extensive human supervision lest they go drastically astray. Anderson's TRIAD system for

the SSL's Beam Assembly Teleoperator (BAT), for example, learns to mimic simple structural assembly tasks by sampling and storing an exemplar operation performed by a human operator. Unfortunately, this is essentially an open loop algorithm; over time, as the machine calibration begins to drift, the performance of the teleoperator rapidly degrades. Similarly, each expert system contains a *static* database of rules to apply in a given situation; there is no way, short of direct reprogramming by a human, to identify and incorporate into this database pertinent new information gained while on orbit. It is this lack of *robustness*, these degradations in performance because of deviations from the ideal operating conditions, which must be addressed before it will become feasible or desirable to make such systems an integral part of the manned space program.

To incorporate true autonomy into machines, there are several very difficult problems which must be solved; pattern recognition, associative recall, and the ability to incorporate and generalize from new experiences are just a few tasks which would be critical to any fully autonomous robot. Despite increases in the speed of the underlying hardware and the proliferation of new algorithms, these problems remain fundamentally unsolved. Paradoxically, these tasks are those which seem easiest for biological systems. Consider even a simple biological system, such as a bumble bee. The bee is capable of recognizing complex patterns (flowers in bloom), landing on an unknown and uneven surface, gathering and loading pollen, adaptively varying its flight strategy on the journey home to account for the new weight and drag distribution, and communicating its find to other bees. The bee knows nothing about edge finding and pattern extraction, yet it recognizes a flower; it knows nothing about aerodynamics, yet it flies; it knows nothing about adaptive control, yet it can stay aloft in very different flight regimes; it knows nothing about linguistics or information theory, yet it can communicate important abstract information to others of its kind. All this, and much more (e.g. self defense, self repair, and reproduction), in a package no bigger than a microprocessor chip! Since current silicon chip technology is beginning to approach the limitations imposed by physics, it is clear that the solutions to the problems in machine autonomy are not in faster, more efficient computers; new insight into the problem is required.

So why, then, can biological systems accomplish these tasks so well? There is a growing school of thought which believes the answer lies in the massive parallelism inherent in the architecture of the neural pathways of living systems. Seen in this context, abilities such as pattern recognition and learning are *emergent* properties of the asynchronous interactions of millions of one bit "processors". Each "processor" makes purely local decisions, based upon the behavior of neighboring processors, about whether to turn on or off, and yet the ensemble of these decisions gives rise to complex forms of

behavior. The whole, according to this theory, is much greater than the sum of its parts. Given that neural time constants are on the order of milliseconds while computer speeds are measured in nanoseconds, the observed architectural differences between biological and man-made computing systems form a very plausible explanation for the above performance discrepancies.

This idea has its roots in over a century of neurology and neural modeling. The general approach has been to develop mathematical models of observed neurophysiological phenomena, then analyze the resulting equations for their "computational" abilities. McCullogh and Pitts (1943; also Landahl *et al.*, 1943) were the first to demonstrate and systematically analyze the computational abilities of networks of model neurons. This analysis was augmented in 1948 by D. O. Hebb (1948) who proposed, based upon then current neurophysiological research and the novel demonstrations of Pavlov (1928), a rule whereby neurons could change the effect they exerted on adjacent neurons in the network, thus laying the foundations for a model of learning. The synthesis of these two seminal approaches was effected by Rosenblatt (1962) who, throughout the late fifties and sixties, spurred research into the Perceptron, a neural model which could exhibit a wide range of learning behavior. In particular he was able to prove the *Perceptron Convergence Theorem* which states that Perceptrons can learn, in finite time, any linearly separable input-output mapping. This research was advanced significantly in 1960 with the development by Widrow and Hoff (1960) of the *delta rule*, a gradient descent method which would ensure that the mapping achieved by the Perceptron was optimal in a least squares sense.

About the same time as the development of the Perceptron, but in an unrelated field, the Russian mathematician Kolmogorov (Lorentz, 1976) proved the very important result that any continuous function of n variables defined on the unit hypercube can be written as the linear superposition of functions of a *single* variable defined on the unit interval. in particular:

$$f(x_1, \cdots, x_n) = \sum_{q=1}^{2n+1} g_q\left(\sum_{p=1}^{n} \Phi_{pq}(x_p)\right) \qquad (1.1)$$

where the Φ_{pq} are continuous, monotonically increasing functions on the interval I = [0,1], and the g_q are continuous. This proof has been refined by several mathematicians (Lorentz, 1962; Sprecher, 1964), with the result that equation (1.1) can be expressed more simply as:

$$f(x_1, \cdots, x_n) = \sum_{q=1}^{2n+1} g\left(\sum_{p=1}^{n} \lambda_p \Phi_q(x_p)\right) \qquad (1.2)$$

where each λ_p is a constant, and the Φ_q satisfy the conditions imposed on the Φ_{pq} stated above. The importance of this theorem to neural modelers will become plain in the next chapter, but put briefly, this result establishes a theoretical justification for the claim that networks of simple, neuron-like processors can compute arbitrarily complex functions of their inputs.

Research into neural models in general, and the Perceptron in particular, was all but completely quenched with the analysis of these models by Minsky and Papert (1969). The two major points of Minsky and Papert's research were that: 1) despite Rosenblatt's convergence theorem, there were certain I/O mappings which could *never* be learned by the Perceptron, notably XOR and parity; and 2.) the number of Perceptrons required to solve a given problem rose faster than exponentially with the size of the problem. Despite this analysis, several researchers pushed neural modeling through the seventies, mostly concentrating on the abilities of *untaught* perceptron-like networks to "free associate" and "compete" and thereby form their own internal representations of the environment to which they were exposed. Grossberg and Kohonen (Grossberg, 1976a, 1976b, 1982, 1987; Hestenes, 1983; Kohonen, 1984) among others, have published fascinating results on these topics, although only Grossberg (1988; Grossberg and Kuperstein, 1986) has demonstrated practical uses for such networks.

In the early eighties, Hopfield led a resurgence of interest in neural architectures by developing and simulating networks of analog elements which could both act as associative memory elements (Hopfield, 1982, 1984) and obtain good solutions to NP complete optimization problems (Hopfield and Tank, 1985; Tank and Hopfield, 1986). However, it was not until 1986 that an answer was found to the some of the criticisms posed by Minsky and Papert. Rummelhart *et al.*(1986b) succeeded in showing that perceptron-like neurons arranged into *multiple layers* could escape at least the first of these criticisms. Their development of the generalized delta rule, or *back propagation* algorithm, for deterministic, multilayered networks has proven capable of learning precisely those mappings of stimuli to responses that the Perceptron could not. At the same time, it was shown (Ackley *et al.*, 1985; Hinton and Sejnowski, 1986) that the flavor of Kirkpatrick's simulated annealing algorithm (Kirkpatrick *et al.*, 1983) could be used to design networks of *stochastic*

neurons, called *Boltzmann machines*, which could also overcome the limitations of the Perceptron. It is not yet known, however, if these new architectures will escape the exponential growth problem.

These new algorithms have produced some spectacular results. The most profound to date owe to Terrence Sejnowski who, using the back propagation algorithm, developed a network simulation which could be taught to read aloud English text (Sejnowski and Rosenberg, 1987). As a true test of the plasticity of these models, Sejnowski then used the same network simulator and trained it to interpret sonar traces better than similarly trained human subjects! (Gorman and Sejnowski, 1988) During these experiments, Sejnowski demonstrated some of the properties of trained neural networks which are drawing interest to the field:

- They can be made to learn *arbitrary* mappings of inputs to outputs.
- They can *generalize* the mappings they learn and produce the correct responses for inputs they have never before encountered.
- They are *robust* to failures in internal network components and can actually *self-repair* damage.
- They can *adapt* to changes in environmental stimuli.
- They can recognize previously learned input patterns even in the presence of additive noise.

Currently these experiments are performed in software on ordinary computers. If these networks could be implemented on silicon or gallium arsenide chips, it is easy to imagine how they could be used to overcome some of the problems associated with spaceborne hardware. Due to the intense radiation, space is a notoriously destructive environment to electronic circuitry; computational elements which exhibit robustness to internal component failure, and even self-repair capability, are certainly worthy of attention. In fact, the Jet Propulsion Laboratory (JPL) has launched a research program into neural networks for precisely these reasons. Further, the ability of these networks to learn, adapt, and generalize would be quite useful in solving the above noted roadblocks to machine intelligence, *if these qualities can be practically extended to nontrivial problems in machine autonomy*. The specter of exponential explosion in network size is still a real possibility which has yet to be convincingly addressed.

Spurred by Sejnowski's demonstrations, researchers have started to examine these issues. Grossberg (1988) and Kuperstein (1988; Grossberg and Kuperstein, 1986) have begun to explore whether neural networks can be used to control robots. In late 1987, the SSL decided to initiate a research program in the possible uses of neural networks in space activities in general, and specifically in telerobotics. This thesis presents the initial results of these investigations.

In the following is examined the possibility of using Rummelhart's back propagation networks to control, or at least regulate, a variety of processes governed by differential equations which are unknown *a priori* to the network. This is the type of low level control problem which would have to be routinely solved by any robotic adaptation scheme. The idea of neural networks which learn to control dynamic systems is not new--it dates at least back to the heyday of the Perceptron (Widrow, 1964; Ku, 1964). In fact, much of this early research established the foundations for some of the modern adaptive control theories. More recently, Barto and Sutton (1982, 1983) have reexamined this idea, with some success. All these earlier schemes suffer from the intrinsic limitations of the Perceptron; however, as this thesis is being prepared, control algorithms using the new neural network techniques are beginning to appear (Guez *et al.*, 1988; Kawato *et al.*, 1987). To the best of the author's knowledge, no studies of learning control have yet been done which utilize back propagation networks, although at least one other such study is in preparation (Showalter, 1988).

Before continuing, it is necessary to draw a very important distinction between the concepts of *learning control* and that of *adaptive control.* Such a distinction is not explicitly drawn in the literature, so a definition is herein proposed, based primarily upon how stability is achieved. *Learning controllers* develop an "intuition" about the process they are to control by experimentation with the plant dynamics; since this experimentation can (and perhaps should, to ensure that the learning is sufficiently rich) drive the process unstable initially, a *trainer* or *teacher* is required to provide critical guidance (i.e. to say when the control and plant response is "good" and when it is "bad"), and to shut down the controller and reset the process to rest conditions when the plant becomes unstable. A learning controller is said to be completely trained when it has developed a control strategy which results in global asymptotic stability to the desired equilibrium (or trajectory) for the plant, without further intervention of the trainer. An apt analogy to a learning controller would be a child learning to walk; the child will fall many times, and be helped back to his or her feet by an adult, before being able to walk unaided. Further, once this knowledge is gained, barring catastrophe, there is no further need for the trainers (adults). *Adaptive controllers*, in contrast, will be considered that set of controllers which contain adjustable

coefficients in their (nonlinear) control laws, for which the parametric adjustment mechanism is sufficient to ensure global asymptotic stability to the desired equilibrium (or trajectory) for the plant *for all time.* That is to say, adaptive controllers must *never* become unstable as they adjust their control laws to accommodate the plant dynamics.

Chapter 2 presents a basic introduction to the mathematics of neural networks, and develops the equations of the Neuromorphic Controller (NMC). Chapter 3 displays the results of simulations in which the NMC learns to construct control laws which regulate the output of a double integrator plant, as well as analyzing in depth the form of the control strategy employed and the impact of each of the parameters in the algorithm on the control laws developed by the network. Chapter 4 presents the results of similar experiments with a much wider range of process dynamics, both linear and nonlinear, using both linear and bang-bang actuators. Finally in Chapter 5, the findings of this thesis are summarized and plans for future research in this area are suggested.

Chapter 2: Theory and Setup

2.1 Neuroanatomical Background

(Stevens, 1966, 1979; Gray, 1977; Adrian, 1980; Kandel and Schwartz, 1985)

Figure 2.1.1 shows a typical neuron. These highly complex cells can be functionally abstracted into their principle components: the cell body or *soma,* the axon, the dendrites, and the synaptic bulbs. The axon and dendrites are the communication channels through which neurons pass information. The *Principle of Dynamic Polarization,* as first observed by Santiago Ramon y Cajal (1933), holds that information flow in networks of neurons is consistent and unidirectional: the dendrites and soma receive inputs from the axons of adjacent nerve cells, a decision is made by the neuron whether it should "fire" or not, and the resulting output is passed out along the axon. Although nerve cells typically have but one axon, it may branch many times with the result that each neuron to which the axon connects receives exactly the same signal.

When a neuron "fires" it does so by generating a +100mV impulse, or *action potential,* which propagates down its axon. Since the magnitude of this action potential is fixed, neurons use frequency encoding to convey their relative degree of excitation; the more stimulus a neuron receives, the more +100mV impulses it outputs per unit time. It takes the axon anywhere from one to two milliseconds to recover from the passage of an action potential (a period of time known as the *refractory period*), so the upper limit to the frequency of this impulse train is about 500-1000 Hz. The frequency of impulses output by a neuron is thus a continuous, monotonically increasing function of its total excitation, which starts at zero and saturates at about 1000 Hz.

The axonic branches connect to the dendrites and soma of the adjacent neurons through the synaptic bulbs. When an action potential arrives at the end of the axon, neurotransmitters are released into the *synaptic cleft* (Figure 2.1.2) which separates the axon half of the synapse (known as the *presynaptic element)* from the dendrite/soma half (the *postsynaptic elements).* This neurotransmitter is then absorbed by the postsynaptic elements where it causes a voltage change. Depending upon the type of neurotransmitter released, the voltage change will either be positive (*depolarization*) or negative (*hyperpolarization)* . The total postsynaptic voltage change is proportional to the rate at which presynaptic action potentials arrive, and hence to the firing frequency of the adjacent neuron. The voltage change at a synapse is thus a *temporal summation* of the impinging impulse train. Synapses can vary in the efficiency by which they convert action potentials

to neurotransmitter and in the type of neurotransmitter emitted; thus the magnitude and sign of the postsynaptic voltage change in response to an impulse train of a given frequency will vary from synapse to synapse.

To a first approximation the soma acts as a summing amplifier. This organelle performs a *spatial summation* of the instantaneous magnitude of the voltage changes seen over all the postsynaptic elements of the cell. If the net voltage change is above a certain threshold, the soma induces an action potential in the root of the axon which is then propagated as described above. If the voltage is below the threshold, the neuron remains quiescent. The initiation of an action potential is thus an "all or nothing" proposition. It is important to note that the response of the neuron is a *nonlinear* function of the stimulus (total voltage change) applied to it. It is precisely this characteristic which makes it possible for networks of neurons to make nontrivial computations.

2.2 Mathematical Neural Models

For biochemical reasons the neuron is limited to a single +100mV voltage spike every 2 msec or so, and thus must employ frequency encoding to convey intensity information. Mathematically, of course, there are no such limitations. The information carried in the neural impulse trains can thus be abstracted by defining a *constant* variable, σ_i, to represent the total neural output. The larger this constant, the larger the excitation of the neuron, and the higher the frequency of the output voltage spikes; thus, physically σ_i represents the time averaged frequency of the impulse train output by a neuron. With this definition, the temporal summation performed at the synapse becomes a simple linear weighting of this incoming constant signal, and the spatial summation performed by the soma is then just a linear summation of all the weighted incoming signals. In this context, networks of neurons can be viewed as analog electric circuits, with the dendrites and axons the "wires" which carry analog voltage levels, and the soma a summing amplifier. This is exactly the view which has inspired Hopfield in his research, with great success.

Figure 2.2.1 shows these ideas schematically. Let a network consist of n neurons, labeled $i = 1,\cdots, n$. Each neuron can receive input from either the environment or from other neurons in the network, and each neuron can connect to any other neuron. For each neuron the *total neural input* is defined as:

$$q_i(t) = \sum_{j=0}^{m} W_{ij}(t)\sigma_j(t) \qquad (2.1)$$

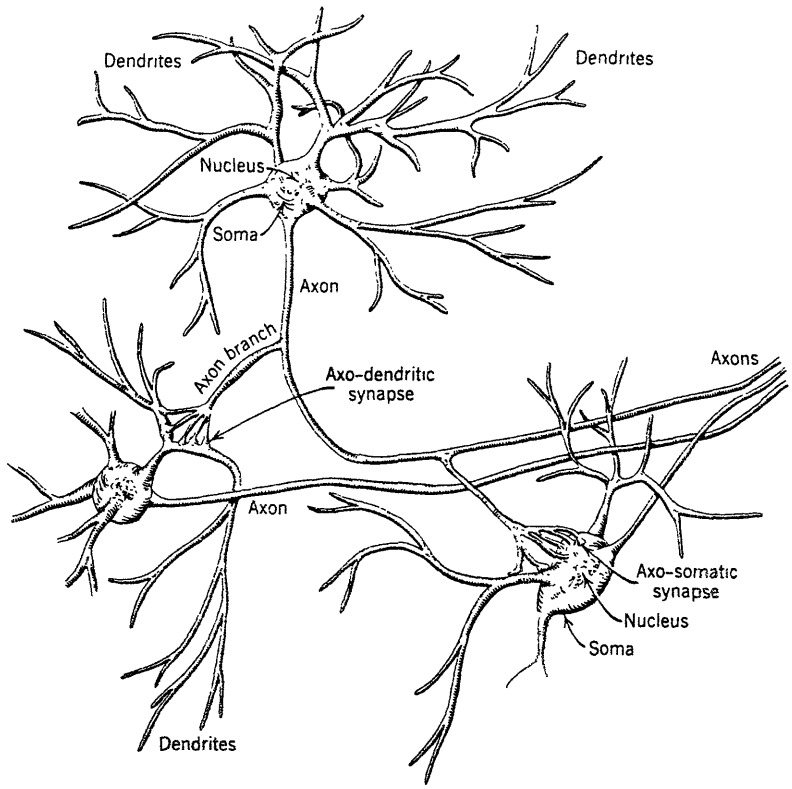

Figure 2.1.1: A typical abstracted neuron. From (Stevens, 1966).

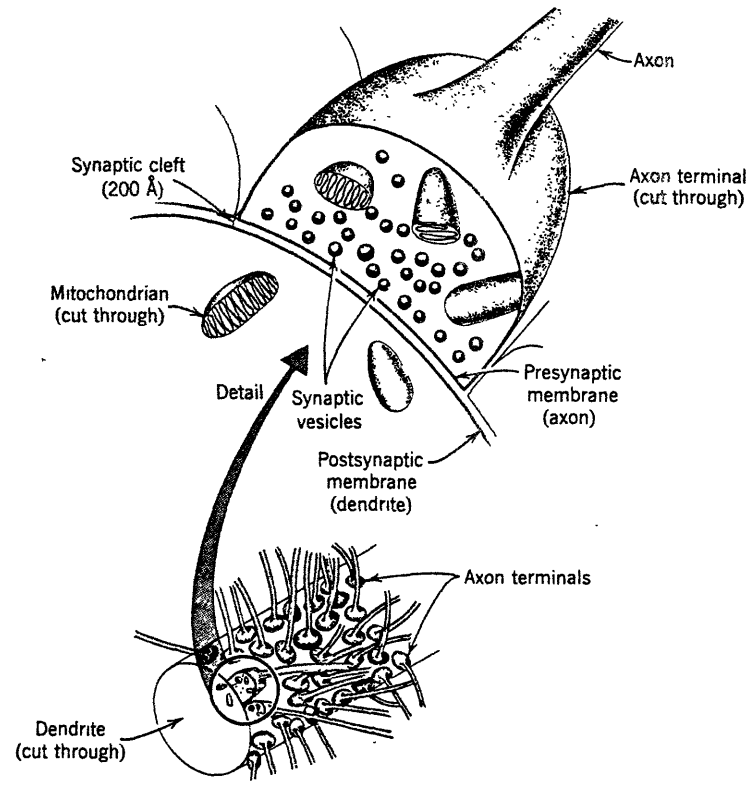

Figure 2.1.2: A typical abstracted synapse. From (Stevens, 1966).

where j varies over the m synapses neuron i makes with adjacent neurons or environmental stimuli, $\sigma_j(t)$ is the presynaptic magnitude of the j input, and W_{ij} is the synaptic weighting of the j input to neuron i.

Neuron i will itself emit a signal based upon its total input. Several different firing models are possible including the binary model originally proposed by McCulloch and Pitts (1948):

$$\sigma_i(t) = f_i(q_i(t)) = \begin{cases} 1 & \text{if } q_i - \theta_i > 0 \\ 0 & \text{if } q_i - \theta_i \leq 0 \end{cases} \tag{2.2}$$

a sigmoidal model:

$$\sigma_i(t) = f_i(q_i(t)) = \frac{1}{1+\exp(k(-q_i(t)+\theta_i))} \tag{2.3}$$

and a linear model:

$$\sigma_i(t) = f_i(q_i(t)) = q_i(t) \tag{2.4}$$

Notice that each neuron in the network may have a different firing law; this distinction will be helpful in the derivation below. The constant term in the sigmoidal model controls the steepness of the sigmoid function, as shown in Figure 2.2.2. The term θ_i in the sigmoid and binary models represents a bias signal, or threshold, internal to the neuron. This bias level can be adjusted just as the synaptic weights. In practice however, it is usually convenient to model the bias as an adjacent neuron which is always on; the process of learning the bias then is the same as learning the synaptic weight to this neuron. While the binary model has been useful in several of the modern algorithms (Grossberg, 1986; (Hopfield, 1982), the differentiability of the linear and sigmoidal models make theoretical analysis more tractable and will hence be used in the algorithm described below. Notice, however, from Figure 2.2.2 that sufficiently large values of k in equation (2.3) will cause the sigmoidal response to approach that of the binary model. All sigmoidal response neurons used in this thesis had values of k = 1.0. Notice as well that the sigmoidal model comes closest to capturing the actual (frequency) output by a neuron, monotonically increasing as a function of increased excitation, possessing upper and lower saturation levels.

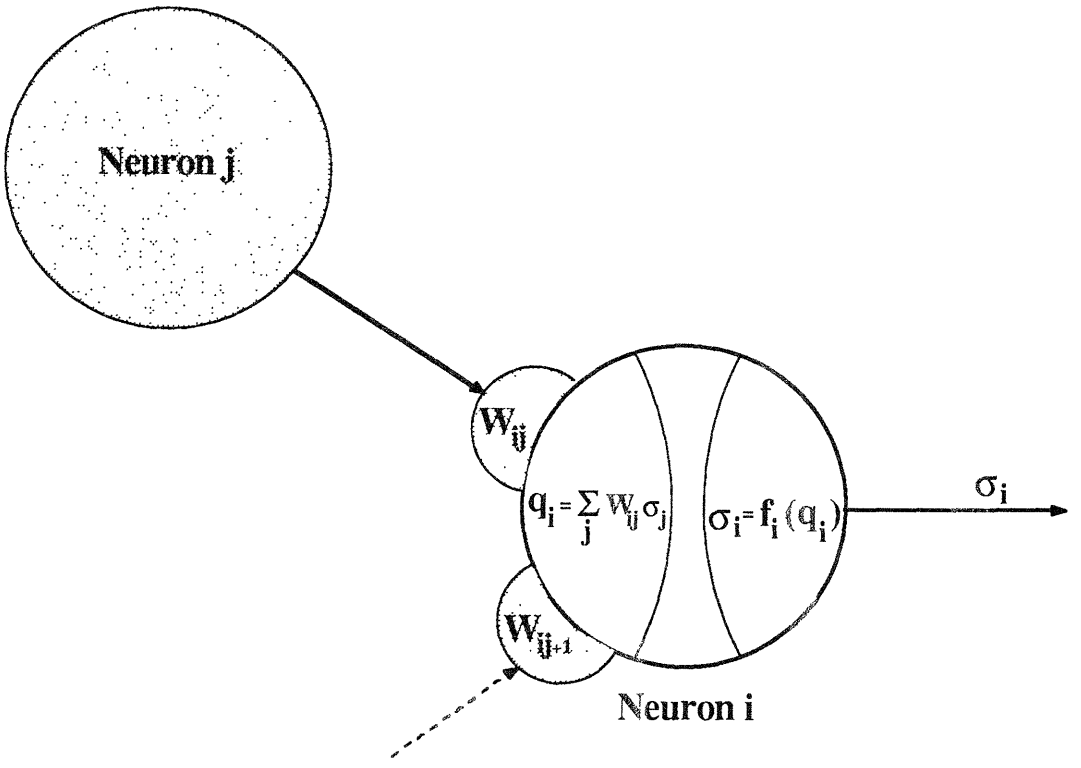

Figure 2.2.1: Mathematical abstraction of biological neural structure. Based on a similar figure in Rummelhart *et al.* (1986a).

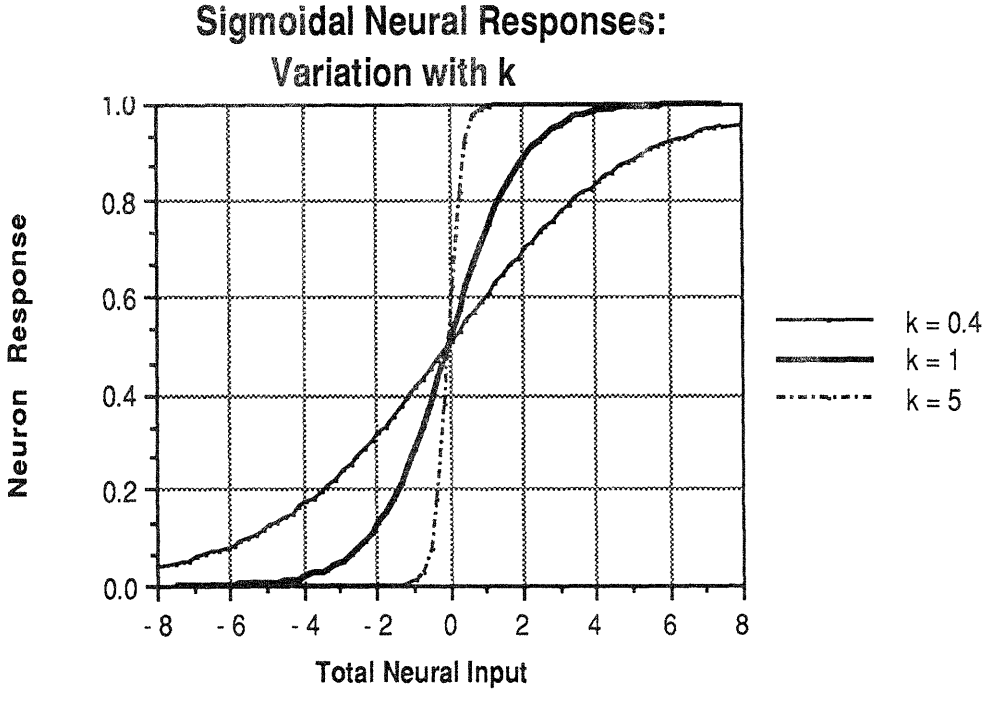

Figure 2.2.2: Sigmoid function variation with k.

2.3 Neural Network Topologies and Learning Models

Figure 2.3.1 shows a typical network of interconnected neurons. There is, in theory, no limit to the number and scope of the neural interconnections; below, however some simplifying assumptions will be made. At any instant in time, information is contained in the network in two distinct forms: the pattern of activity across the individual neurons (the $\sigma_i(t)$), and the values of the synaptic connection strengths, $W_{ij}(t)$. The $\sigma_i(t)$ typically vary tremendously with time, while the $W_{ij}(t)$ vary much more slowly. In analogy with observed human psychophysiology these are sometimes respectively referred to as the short term memory (STM) and long term memory (LTM) traces (Hestenes, 1983). It have has already been shown how the former changes with time, in this section an algorithm for updating the connection strengths is examined.

The most common model for modifying synaptic strengths arises from the work of Hebb in 1949. Hebb's idea was that synaptic efficiency would change as a function of the correlation of pre- and post-synaptic signal strengths. This hypothesis has been recently substantiated in neurobiological experiments (e.g. Castellucci and Kandel, 1976), and forms the core of most current neural network algorithms. Mathematically, this idea can be expressed as:

$$\Delta W_{ij}(t) = \eta \sigma_i(t) \sigma_j(t) \tag{2.5}$$

where i and j are adjacent neurons and η is the learning rate. In practice it is usually desirable to add a "momentum" term to this equation (Sejnowski and Rosenberg, 1987):

$$\Delta W_{ij}(t) = \eta \sigma_i(t) \sigma_j(t) + \alpha \Delta W_{ij}(t-dt) \tag{2.6}$$

Consider now a subset of the above network topology, shown in Figure 2.3.2. The network is arranged into several layers: an input layer which receives only signals from the environment, an output layer which emits signals into the environment, and one or more hidden layers which the network can use to encode environmental information and develop sophisticated I/O mappings. In practice only one hidden layer is needed; it can be shown that three layers (input, output, and hidden) of nonlinear neurons are sufficient to allow the network to develop arbitrarily complex mappings from input to output spaces (Lippman, 1987).

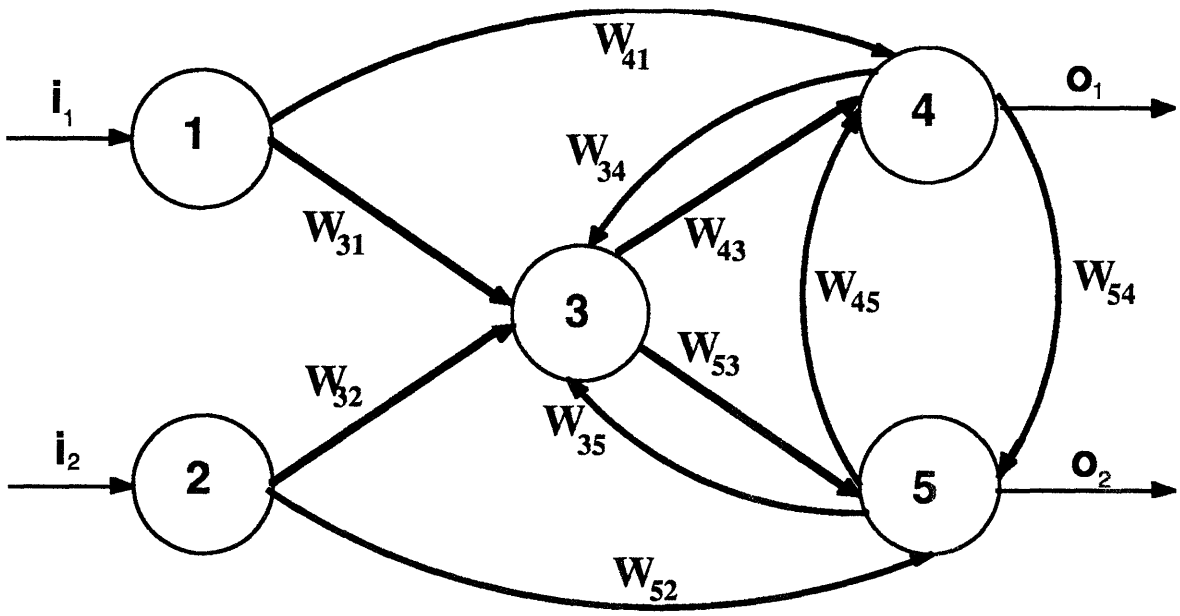

Figure 2.3.1: An example of an artificial neural system (neural network)

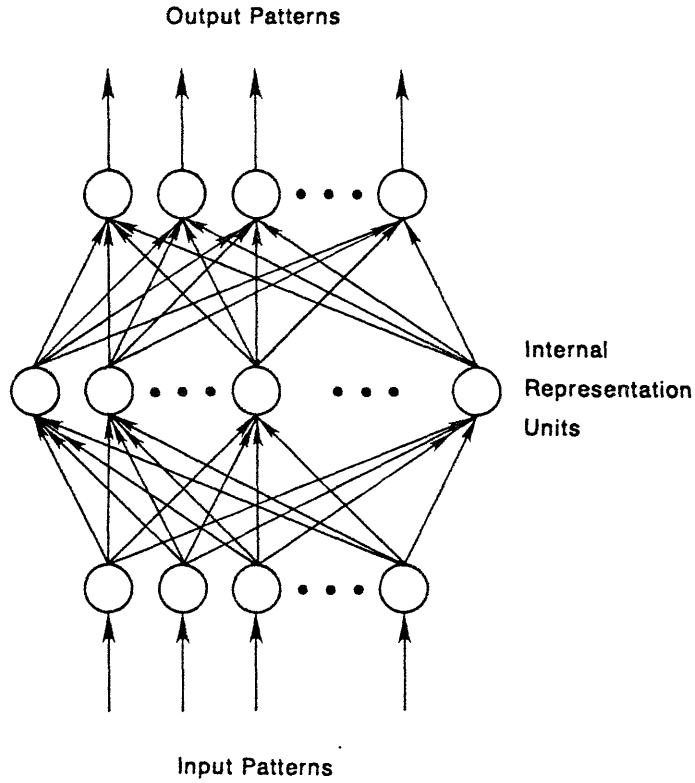

Figure 2.3.2: Back propagation network topology. From Rummelhart *et al.*(1986b)

The type of network shown in Figure 2.3.2 is referred to in the literature as a *layered, feedforward* network. Each layer communicates only with successive layers; there is no feedback within the network either between layers or between individual neurons, nor can neurons communicate with other neurons in the same layer. When a pattern of environmental stimuli is "presented" to the neurons of the input layer by clamping the output of these neurons to environmental determined values, the output is then computed by calculating, synchronously by layer, the response of the neurons in each successive layer. Neural activity thus proceeds in a wave from the environment, to the input layer, through the hidden layers, to the output layer.

The recent developments due to Rummelhart *et al.* (1986b) have shown that the network illustrated in Figure 2.3.2 can be made to learn arbitrary I/O mappings. Define a desired vector of outputs $t_i(t)$ which one requires the network to produce at the output layer in response to an environmental stimulus vector, $s_j(t)$, applied to the input layer. Here the subscript i ranges over the set of output neurons, $i = 1, \cdots, m$, while j ranges over the set of input neurons, $j = 1, \cdots, n$. Define the output error vector, $\delta_i(t)$, to consist of the current deviation of each neuron of the output layer from the output desired for that neuron, weighted by the derivative of its neural activation function; that is,

$$\delta_i(t) = f_i'(q_i(t))(t_i(t)-\sigma_i(t)) \tag{2.7}$$

where, again, i ranges over the output neurons. This error can now be *back propagated* through the network to define an equivalent error at each neuron:

$$\delta_i(t) = f_i'(q_i(t))\sum_k \delta_k(t)W_{ki}(t) \tag{2.8}$$

where $f_i'(q(t))$ is the derivative of the activation function of neuron i, and k varies over the set of neurons to which the axons of neuron i connect. This process effectively solves the credit assignment problem: i.e., given that the output is currently incorrect, to what extent does the activity of adjacent neurons contribute to the wrong decisions made at the output neurons. The back propagated error gives a numerical evaluation of this criterion.

With this formula for the "generalized error", the weight change at each time step which will cause the actual and desired output vectors to converge is then given by (Rummelhart *et al.*, 1986b):

$$\Delta W_{ij}(t) = \eta\sigma_i(t)\delta_j(t)+\alpha\Delta W_{ij}(t-dt) \tag{2.9}$$

The proof hinges on the creation of a metric, E, of the deviation of the network from the ideal state; it can then be shown that the weight change formula (2.9) results in $\Delta E \leq 0$, and hence implements a gradient descent search for the ideal weights.

It is clear, now, how Kolmogorov's theorem (equation 1.2) lends credence to these results. Each neuron in a back propagation network is computing a single, monotonically increasing function of its total net input which, for sigmoidal neurons, is restricted to the unit interval I = [0,1]. The outputs of the network are a function of linear superpositions of each of these responses. The results of Kolmogorov's theorem thus claim that such a network, properly arranged, should be able to compute any function of the n input variables; back propagation is then just a technique for iteratively approximating the weights required for each of the single valued functions in (1.2) which will cause the network to reproduce the desired n dimensional function.

For most back propagation problems there are a fixed number of input-output patterns which must be learned by the network, For these networks, t takes on discrete values t = 1, ···, kN where k is the number of patterns to which the network must learn to respond (sometimes also called the *training set*), and N is the total number of presentations of the training set which is made to the network as it is being taught. For example, in the XOR problem there are 2 inputs, $\mathbf{s}(t) \in \Re^2$, and one output, $\mathbf{t}(t) \in \Re^1$, and there are a total of four patterns in the training set. Rummelhart's initial experiment required 558 presentations of the XOR training set to his network until it had learned the mapping, so for this experiment t varied as t = 1, ···, 2232. The synaptic update formulae are thus finite difference equations with unit delay.

It has been noted above that this weight updating rule essentially implements a gradient descent in the solution of the mapping problem described above. As such it is possible for the network to become stuck in local minima of the weight space and hence fail to achieve a solution. Despite these possible limitations, networks employing this teaching technique have been taught to perform such varied tasks as pronouncing English from written text (Sejnowski and Rosenberg, 1987), mimicking logic operations (XOR, left and right shift, parity checking, etc), performing mathematical operations, and sequencing actions (Rummelhart *et al.*, 1986b), and even interpreting sonar traces (Gorman and Sejnowski, 1988). On very few occasions has the algorithm been observed to fail to achieve the desired mapping by becoming stuck in a local minimum of the solution space.

The concise, mathematical nature of the above models may give the reader a false sense of security regarding the extent to which the biological processes of thinking, learning, and adaptation have been neatly encapsulated. It should be emphatically stated that these abstractions are, most probably, *not* how living organisms actually process

information, nor is there any claim to this effect attached to the above equations. Far too much of known neurophysiology has been completely ignored, and far more still remains unknown about how biological neural systems actually function. Neural time delays, morphological network changes, synaptogenesis, and dendritic shunting, to name but a few of the known features, have all been ignored in the model discussed. Taken on its own merits, however, this neurally *inspired* model has shown itself to be worthy of attention, as the recent results cited demonstrate.

Nor should the above network architecture or neural models be considered the only candidates for serious study. Precisely because so much of known neurophysiology has been neglected in this algorithm, and also because of the harsh restrictions back propagation places on the allowable network topologies and neural timing, many different models have flourished, several employing non-Hebbian learning schemes; many of these, as briefly discussed in the Introduction, have had great successes in their own right. In the following derivation, however, back propagation networks are used because they have been *proven* to at least implement a gradient descent search for the solution to the mapping problem posed, and further because they make use of a structured input-output format which is ideal from a control theoretical standpoint.

2.4 The Neuromorphic Control Algorithm

Feedback control of arbitrary process dynamics is essentially a mapping procedure. What, if any, weighted combination of the current states of the process should be fedback as control signals to force the plant to an arbitrary final state, subject to certain performance criteria? In the previous section an architecture using neural dynamic models was examined which was capable of learning arbitrary mappings of input signals to output signals. In this section, an architecture is described which attempts to harness this property for the real time generation of feedback control laws in the absence of *a priori* knowledge of the process dynamics.

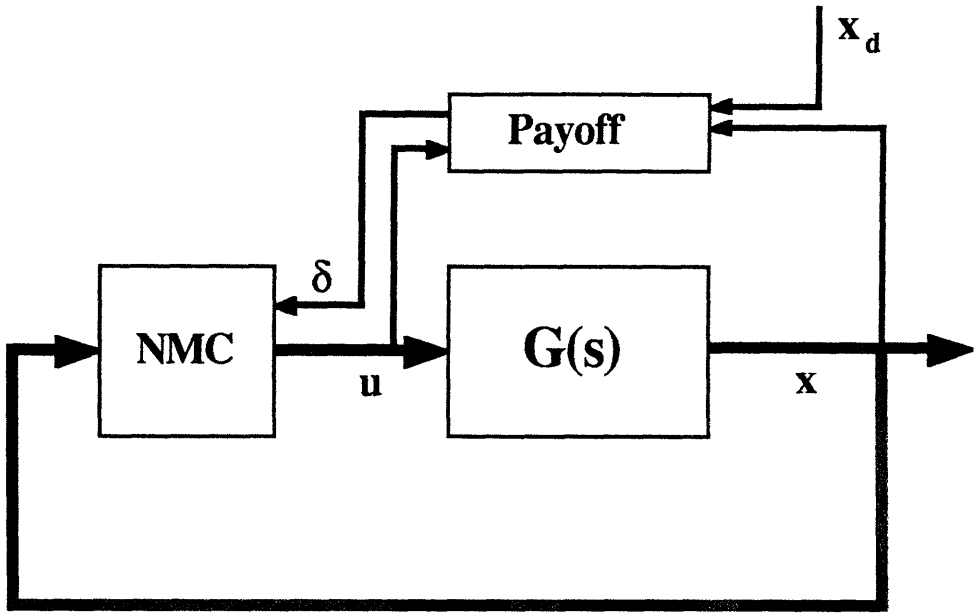

Figure 2.4.1: Block diagram of the neuromorphic controller

Figure 2.4.1 shows the general structure of the neuromorphic controller (NMC). The NMC is a feedforward, multilayered, artificial neural system of the type described in Section 2.3. The input and output layer neurons each have linear activation functions, equation (2.4), so they can more easily encode the full range of analog signals needed to interact with the continuous dynamics G(s). The hidden layer neurons all have sigmoidal activation functions, equation (2.3), and only the hidden and output neurons have biases. The feedforward connections are complete, so that each neuron of one layer makes connections with every neuron in each of the following following layers.

For generality the NMC notation is developed for multi-input, multi-output (MIMO) systems, although this thesis will examine only single-input, single-output (SISO) applications in detail. Work is currently in progress to demonstrate the feasibility of the algorithm for MIMO systems. As shown in the figure, at each instant in time the current n dimensional state vector, $\mathbf{x}(t)$, of the plant dynamics, G(s), is applied to the input layer of the NMC system. G(s) is an n by m transfer function matrix which describes the impact of the m controls on the n states. The input vector is propagated through the network layer by layer as described above, and the NMC develops signals, $\mathbf{u}(t)$, at its output layer in response; $\mathbf{u}(t)$ is then fed to the plant as the control input. In this thesis this structure is considered only as a state regulator, with the desired regulator setpoint specified by $\mathbf{x}_d$.

Two crucial assumptions are now made. First it is assumed that the dynamics of the individual neurons are very fast compared to the dynamics of the process to be

controlled; the output of the network thus develops "instantly" in response to the applied stimulus (the plant state vector). This is not an unreasonable assumption since, if the currently proposed VLSI neural processors are used to implement the controller, these have settling times on the order of nanoseconds (Jackel *et al.*, 1986; Alspector and Allen, 1987), and if the neural equations are approximated in software, the update is limited only by the speed of the hardware and software, typically as high as 100 Hz. Second, it assumed that the actual process of updating the synaptic efficiencies is not a continuous process but instead occurs at discrete intervals, although still with a frequency faster than the (expected) process dynamics. While it is perfectly possible to develop continuous time versions of the synaptic update equations (e.g., Cohen and Grossberg, 1986), leaving these in the difference equation form developed above allows the NMC algorithm to be implemented as a digital control system using conventional microprocessor hardware.

The task of the NMC is to construct a control signal, $\mathbf{u}(t)$, as a function of the current plant state $\mathbf{x}(t)$ which will drive the dynamics to the desired final state $\mathbf{x}_d$. Note several problems which instantly arise in trying to apply classical back propagation to this problem. First, back propagation usually works with a finite training set, as described above, but a dynamic system yields an infinity of points in state space which must be mapped to stabilizing control signals. Fortunately, many linear and nonlinear plants can be stabilized with a particularly simple control law, $\mathbf{u}(t) = -\mathbf{K}\mathbf{x}(t)$, where $\mathbf{K}$ is a constant gain matrix. Of course, not just any set of gains will stabilize the system, and it is not even clear if back propagation will allow NMC to discover this underlying simplification of the mapping problem.

The second problem in applying back propagation to the control problem is that back propagation requires an "omniscient teacher" which can show the network exactly what the required outputs are given the current inputs. For the neuromorphic controller this is equivalent to specifying an "ideal" control signal $\mathbf{u}^+(t)$ (presumably computed with exact knowledge of the plant dynamics) which could be used to determine the error signal at the output neurons, i.e. $\delta_i(t) = u_i^+(t)-u_i(t)$. In fact this is exactly the approach used at Stanford in the early 1960's by Widrow and Smith (1964) where a perceptron-like neuron (the MADALINE) was taught the phase plane switching logic for the optimal bang-bang control of a harmonic oscillator. However, for the NMC algorithm implemented here it is assumed that neither the network *nor the teacher* have any *a priori* knowledge of the plant dynamics; thus, construction of an ideal, model control signal is impossible. At best the teacher can observe the output of the plant and somehow communicate to the network when its response is "good" and when it is "bad".

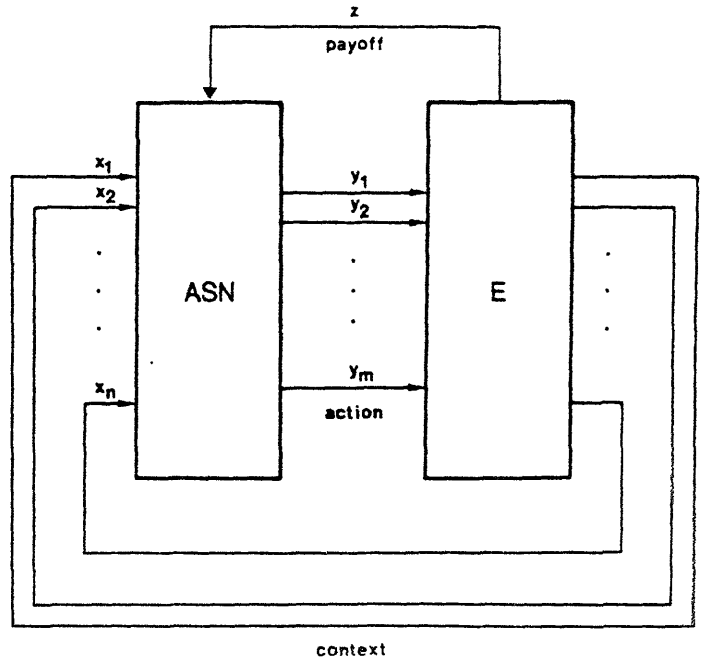

Figure 2.4.2: Schematic of Barto and Suttons's environmental payoff idea. From (Barto, Sutton, and Brouwer, 1981)

What has just been described is the concept of "learning with a critic" or *bootstrap adaptation* first explored by Widrow, Gupta, and Maitra (1973) in the early 1970's. Widrow *et al.* used this technique to successfully teach a perceptron-like network to play blackjack with near optimal strategy. More recently, Barto and Sutton (1981a) have applied a variant of this idea to several problems, including navigation in an "olfactory" gradient field (1981b) and the bang-bang control of an inverted pendulum on a cart (1983), using their perceptron-derived "associative search element". Barto and Sutton had some success with this latter experiment, but their approach has three fundamental drawbacks: first, they required a front end device to artificially partition the state space into 162 separate "regions"; second, they were not concerned with the quality of the solution obtained by the network, only that the pendulum remained within 12 degrees of the vertical for a certain number of time steps; and third, their network did not always evolve a controller which stabilized the system for all time.

The difference between learning with a teacher and learning with a critic lies in the type of feedback which can be provided to the network. A teacher can give specific, exact information as to how and where the output is incorrect; a critic can offer at best qualitative, sometimes even incorrect, feedback. An extremely valuable approach to the implementation of these bootstrap adaptation networks was developed in the earlier research of Barto and Sutton, and is diagrammed in Figure 2.4.2. Their idea is to define a "payoff" function which provides to the network some numerical evaluation of how appropriate its outputs are at each step. A positive payoff signal indicates "reward" or reinforcement, i.e. the synaptic strengths which led to the current decision are valuable and

should be strengthened; similarly, a negative payoff signal indicates "punishment" or inhibition, and indicates that the synaptic strengths involved in that decision should be weakened.

The NMC algorithm used in this thesis essentially attempts a synthesis of these three ideas: back propagation, bootstrap adaptation, and environmental payoff functions. In place of the deviation of the network output from a hypothetical "ideal" control signal, discussed above, the NMC uses a payoff function which is some measure of how far the plant deviates from its desired final state. This payoff function is used as the error signal at the output nodes of the network; the error is then back propagated through the network, as discussed above, using equation (2.8), and the synaptic weights updated, using equation (2.9). If the NMC is successful, the magnitude of the error (payoff) signal will be driven to zero in finite time.

Two different forms of the payoff function have been investigated. The first takes inspiration from Linear Quadratic Regulator (LQR) design. In this *state weighted* form of the payoff, the error vector, $\delta(t)$, is defined as a weighted function of both the deviation of the plant state vector from the desired state and the deviation of the control from some desired control level:

$$\delta(t) = \begin{pmatrix} \mathbf{M_x} & \mathbf{M_u} \end{pmatrix} \begin{pmatrix} \mathbf{x_d - x} \\ \mathbf{u^*} \end{pmatrix} \tag{2.10}$$

Here **M** is the state and control weighting matrix, partitioned for clarity, and $\mathbf{u^*}$ is a normalized control signal. The back propagation formulae always adjust the synaptic weights so as to attempt to drive the error seen at the output terminals to zero. Since many applications require nonzero steady state control to be applied, and since the magnitude of the applied control can fluctuate greatly, simply weighting the control signal in the formulation of the teaching stimulus will drive the NMC unstable; it will be unable to solve the problem as posed. The solution is to define a certain amount of allowable control, u_{ult}, and generate a normalized error vector $\mathbf{u^*}$ such that:

$$u^*_i = -\mathrm{sgn}(u_i)\left|\frac{u_i}{u_{ult}}\right|^n \tag{2.11}$$

n is chosen so that the gradient of this contribution to the error is sufficiently steep for values approaching u_{ult}; a value of n=4 was used for many of the simulations, although in some of the tests conducted this was an experimental variable.

The second approach takes its inspiration from model-reference adaptive control theory, and hence is referred to as the *model reference* form of the payoff. In this formulation, a model trajectory for one or more of the states is defined, and the error signal is computed from:

$$\delta(t) = (\mathbf{M_x} \ \mathbf{M_u}) \begin{pmatrix} \mathbf{x_d(t)-x(t)} \\ \mathbf{u^*} \end{pmatrix} \tag{2.12}$$

The chief distinction here is that the desired plant state is now a continuous function of time, instead of a constant value as was the case in equation (2.10). Notice that even in the model reference approach a term proportional to the normalized control is included; the reasons for this will be discussed below.

Notice particularly that in neither of these formulations does the NMC have any intrinsic information about the dynamics contained in G(s); any control algorithm formed by the controller will hence be devised using information acquired while on-line.

2.5 The NMC Simulator

A simulator was constructed to implement this algorithm. The simulator was initially written and debugged on a Macintosh SE computer under the LightSpeed C compiler, then subsequently ported to Microsoft C version 5.0 running on an IBM PC-AT with 80287 math coprocessor support Construction of the simulator proceeded in two separate phases to ensure the accuracy of the final results. The first phase involved the construction and validation of a neural network simulator. Expecting that this simulation would be used as the basis of future lab experimentation with neural networks, every effort was made to keep the simulator as flexible as possible. Thus, an object oriented programming style was adopted. This approach allows users to define different "flavors" of neurons and networks, but still employ exactly the same high level syntax to manipulate these elements. An explanation of the structures and functions used to implement these features is given in Appendix A, along with the C source code. For the purposes of the experiments described below, two classes of neurons were created, linear and sigmoidal, and one network class, back propagation.

To test the accuracy of the network simulator, the results of Rummelhart's XOR experiment, using his minimal network representation, were first verified . The results obtained after 750 presentations of the training set (kN = 3000), with $\eta = 0.5$ and $\alpha = 0.8$, are shown in Figure 2.5.1(a) compared with Rummelhart's results in 2.5.1(b). The numbers inside each neuron represent the bias level, θ_i, for that neuron. The structure of the synaptic strengths in these two networks is clearly identical, although the absolute magnitudes of the weights observed in this more recent experiment are greater because the network was allowed to learn longer than in the original paper. The astute reader may note that for neither of the networks shown in Figure 2.5.1 is the output ever exactly one or zero in response to the inputs in the training set. Because the sigmoid only approaches these values asymptotically, values of output greater than 0.9 are commonly considered to be 1.0 or "on", while values less than 0.1 are considered to be 0.0 or "off".

Further tests of the simulation facility created for this thesis verified other results cited in Rummelhart's paper, including a network which implemented a three bit shift register and one which implemented binary addition. Several further experiments based upon suggestions imbedded in Rummelhart's paper were conducted; these verified the concept of using linear response neurons in the network formulation, so as to learn *continuous* (as opposed to binary) mappings from the input to the output layers of the network. This last experiment was the first indication that the NMC concept might be viable.

Satisfied that the network simulator worked, it was then combined with a dynamic system simulation package. A fourth order Runge-Kutta algorithm was chosen for this simulator, and since the output would have to be in tabular form for later analysis, a fixed stepsize version of this algorithm was implemented. Dynamic systems were specified to the simulator in state space form through a user supplied subroutine, as discussed in Appendix A. The two simulators were connected by passing the output from the neural network at each time step to the dynamic system simulator as the control signal. The response of the dynamic system (or *plant* in control theory terminology) as a result of its current state and the applied control is then computed, and returned to the network simulator as the network inputs, thus implementing the structure detailed in Figure 2.4.1.

In the following discussions, the *teacher* or *trainer* will be referred to as that part of the network simulator which observes the output of the network and plant and issues criticism using the payoff function. When the output of the plant or network begin to exceed certain predetermined absolute bounds (discussed below), the trainer must also stop the simulation by resetting the plant to its initial conditions, then restart the simulator.

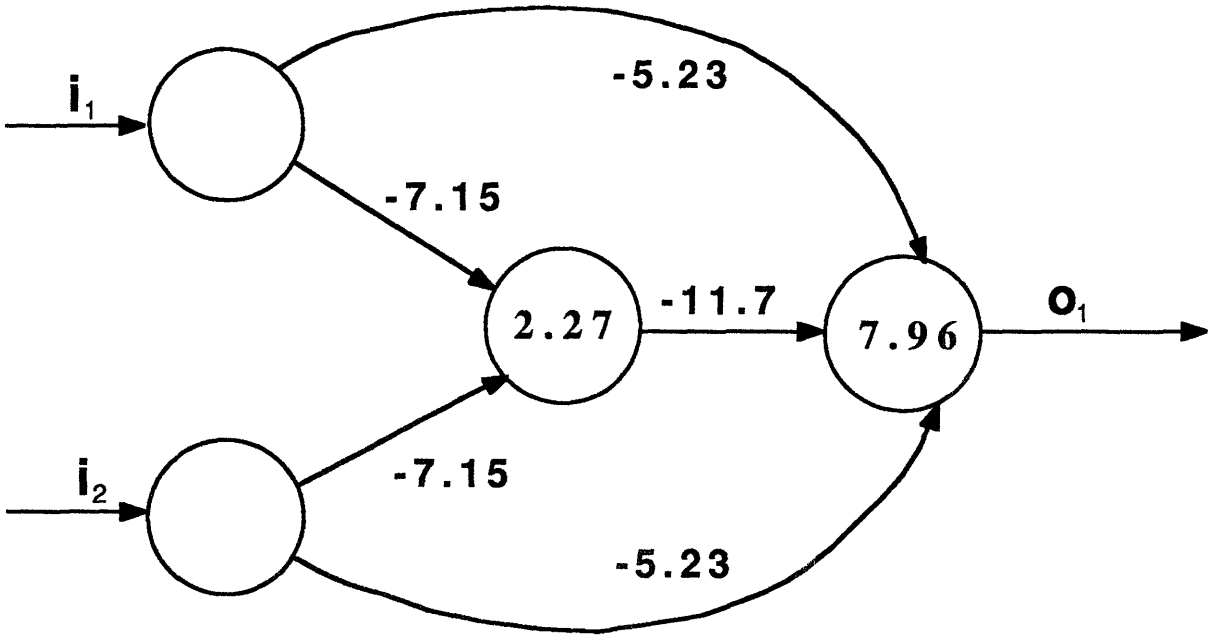

Figure 2.5.1(a): Structure of the observed network which implements XOR

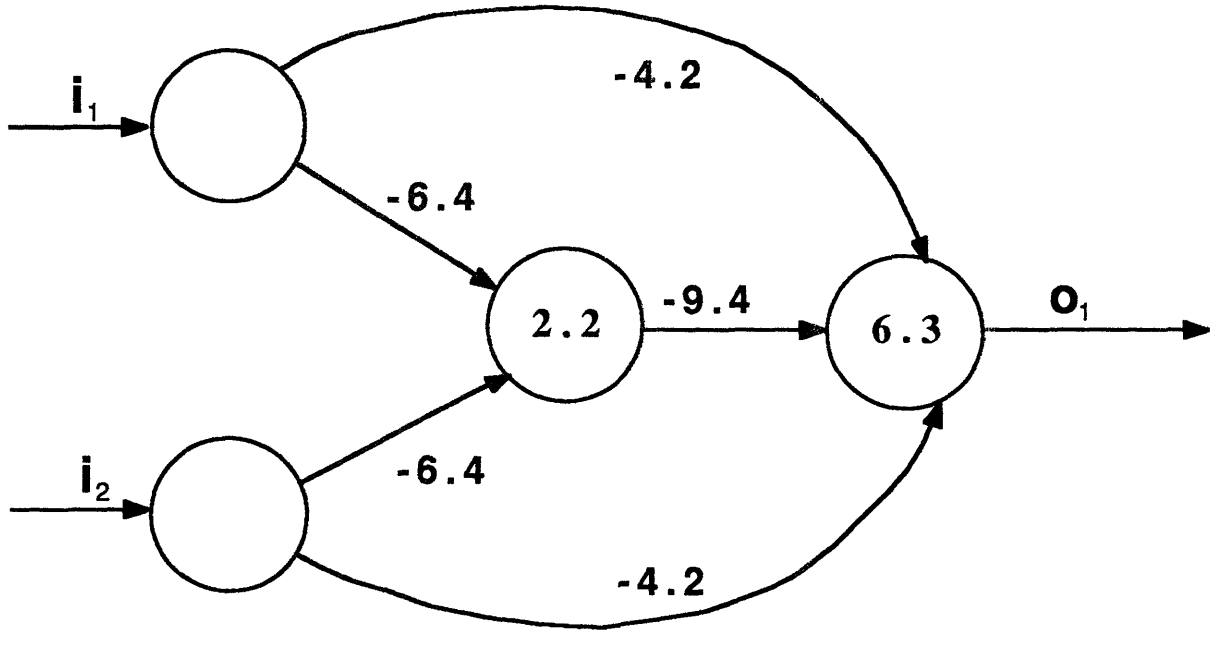

Figure 2.5.1(b): Structure of network reported in Rummelhart *et al.* (1986) implementing XOR

2.6 Implementation of the Algorithm

For the first tests of this algorithm, the NMC's ability to regulate the output of second order, and a few third order, (SISO) systems was examined. With these simplifications, the NMC structure of Figure 2.4.1 reduces to that shown in Figure 2.6.1. The smaller circles represent bias neurons, and the larger circles are neurons whose firing laws are indicated by the letters contained within: L represents linear response neurons and S represents a sigmoidal response neuron; synapses are indicated by the directed arrows. The extension of this topology for third order systems is quite straightforward, as shown in Figure 2.6.2. Notice there are three neurons in the hidden layer for this particular topology. It is not known how to predict the number of hidden neurons required for a given system; as a general rule of thumb at least as many hidden neurons as the dimension of the plant state vector were used.

The particular structure shown in Figure 2.6.1 for second order systems was devised after giving consideration to the minimum size of the network which would be required to implement a control law capable of stabilizing a linear second order plant about a nonzero final state. This minimum network is diagrammed in Figure 2.6.3. If this network could learn the weights $W_1 = -k_1$, $W_2 = -k_2$, and $W_3 = k_1 x_d$, with k_1 and k_2 positive, the resulting control law would be:

$$u(t) = k_1(x_d - x) - k_2\dot{x} \tag{2.13}$$

which will force the system to the final state $\mathbf{x}^T = [x_d\ 0]$, provided the gains k_1 and k_2 are chosen properly. In fact, experiments have shown that this structure is *not* capable of learning the required weights using the NMC algorithm as described above. In a sense this is not surprising, since the weight distribution discussed above is the only possible way to stabilize the system: the network has no degrees of freedom. If the network does not find the correct distribution of weights relatively quickly (before the teacher decides that the network has "failed" and signals a restart), it will probably never find it. The solution to this problem seems to lie in providing the network with at least one layer of hidden, sigmoidal neurons. The sigmoidal properties of these neurons stabilize the learning process itself in addition to providing the network with many additional degrees of freedom to use in designing a controller. Simply adding, arbitrarily, more linear neurons or more layers will not be sufficient, since it is easily shown that multiple layers of linear neurons are equivalent to a single layer with appropriately selected weights. The success of the algorithm seems to hinge upon the nonlinearity of these hidden neurons; indeed

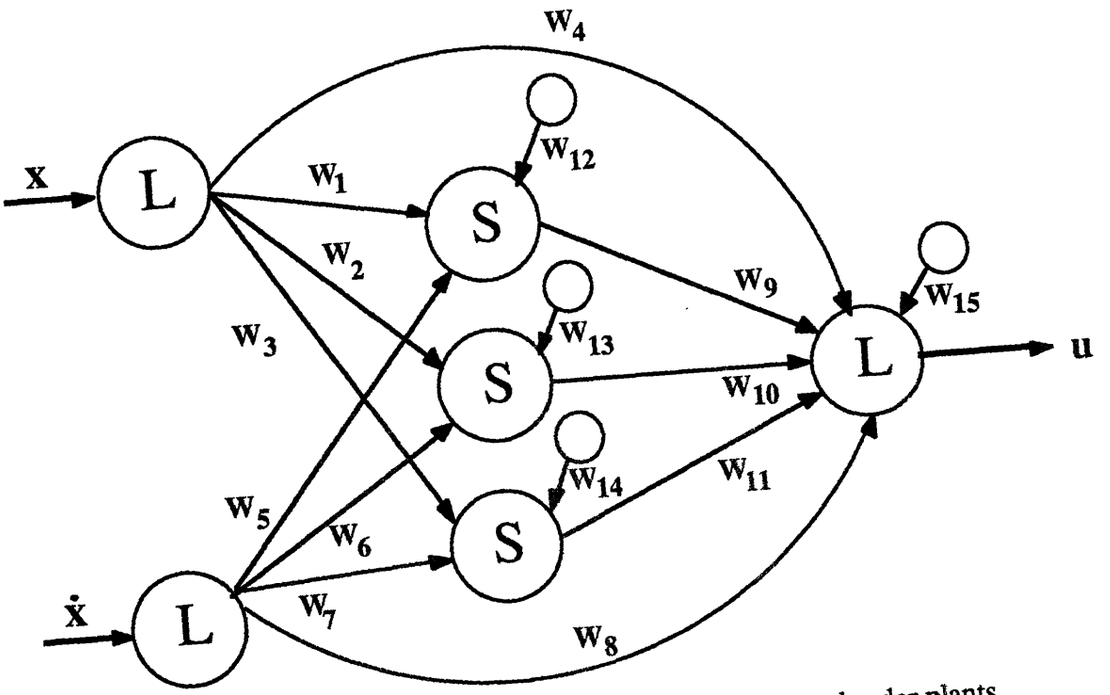

Figure 2.6.1: Structure of the NMC network for second order plants

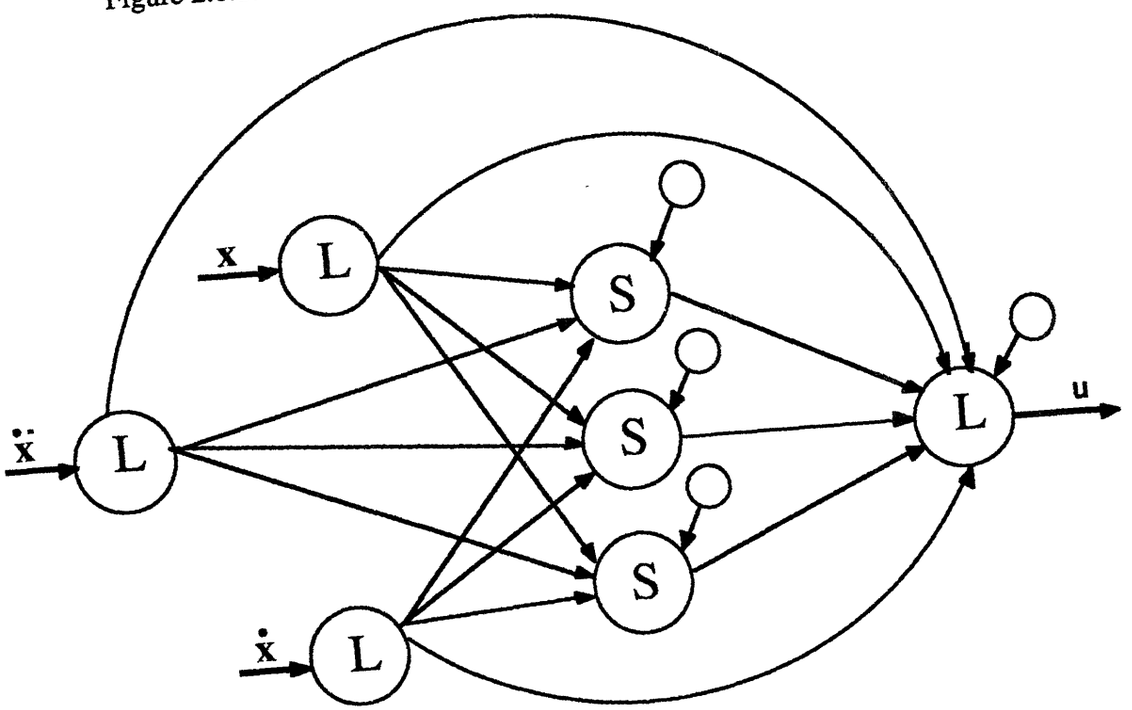

Figure 2.6.2: Structure of the NMC network for third order plants

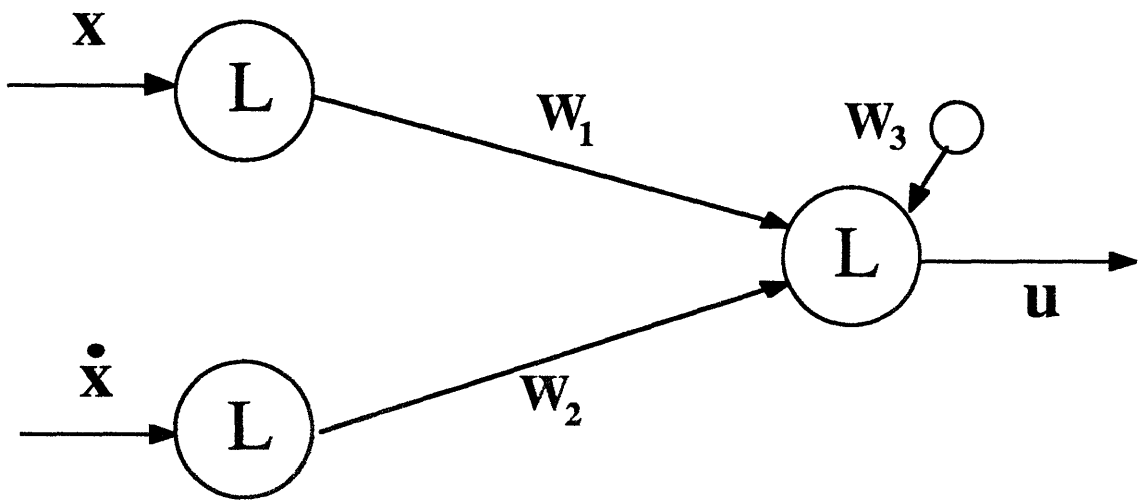

Figure 2.6.3: Minimal "network" structure needed to implement feedback control.

experiments have shown that if the sigmoidal neurons in Figure 2.6.1 are replaced with linear neurons, the NMC algorithm goes rapidly unstable.

For a second order plant and the network topology shown in Figure 2.6.1, the resulting set of neuromorphic controller equations, from (2.1), (2.3), (2.4), (2.8), (2.9), and (2.10) are summarized in Figure 2.6.4 The numbering scheme used in Figure 2.6.1 is repeated in the equations; thus neurons one and two are the linear input neurons, neurons three, four, and five are the hidden neurons, and neuron six is the linear output neuron--neurons seven through ten are the bias neurons. Synapses one through four come from the position input neuron, synapses five through eight come from the velocity input neuron, and synapses nine through eleven come, respectively, from each of the hidden neurons; synapses twelve through fifteen come from the bias neurons and hence represent the threshold values of the hidden and control neurons.

The synaptic weightings were initially set to random values on the interval [_0.3, 0.3] as suggested by Sejnowski and Rosenberg (1987). Each simulation run consisted of two phases, each lasting 10-20 seconds of simulated time; at the beginning of each of these phases the plant was reset to zero initial conditions, i.e. $\mathbf{x}_o^T = \mathbf{0}^T$. In the first phase the network was run with the teacher active; at certain intervals, 1/f, chosen to be an integral multiple of the simulation time step Δt, i.e. $1/f = k\Delta t$, the payoff was computed and the synaptic weights were modified by the above procedures. The rate, f, at which the synaptic weights were changed, and the parameters α and η in equation (2.9) above were experimental variables, although most of the simulation runs used $\alpha = 0.25$ and $\eta = 0.5$, and a synaptic update rate of f = 20 Hz. After a training run, the simulation was repeated with the teacher off-line to determine how well the network was capable of controlling the

process on its own. This two phase process was iterated, usually fifty times, subsequent iterations beginning with the synaptic strengths learned in the previous iterations. Each simulation was conducted with an integration time step of $\Delta t = 0.005$ seconds.

A failure by the network was signalled by the teacher if at any time during training the response exceeded ten times the desired response, or if the control signals issued exceeded five times the maximum allowable control. If a failure occurred, the plant was reset to zero initial conditions, and the simulation begun anew with the previously learned synaptic strengths. A network will be considered to have successfully "solved" the control problem if, after a finite training period, it has developed a pattern of synaptic weights that cause the plant to stabilize about the desired final state without further intervention by the teacher.

Second Order NMC Equations

At each time step (Δt seconds):

- Present input vector to network:

$$\sigma_1 = x \qquad \sigma_2 = \dot{x}$$

- Compute hidden neuron responses:

$$\sigma_3 = sig(W_1\sigma_1 + W_5\sigma_2 + W_{12})$$
$$\sigma_4 = sig(W_2\sigma_1 + W_6\sigma_2 + W_{13})$$
$$\sigma_5 = sig(W_3\sigma_1 + W_7\sigma_2 + W_{14})$$

- Compute control:

$$u = W_{15} + W_4\sigma_1 + W_8\sigma_2 + W_9\sigma_3 + W_{10}\sigma_3 + W_{11}\sigma_4$$

- Integrate plant forward in time:

$$\dot{x} = g(x, u)$$

Every $1/(f\Delta t)$ time steps:

- Compute payoff function:

$$\delta_6 = M_x(x_d - x) + M_{\dot{x}}(\dot{x}_d - \dot{x}) - M_u \mathrm{sgn}(u)\left|\frac{u}{u_{ult}}\right|^n$$

- Back Propagate payoff signal:

$$\delta_3 = \sigma_3(1.0 - \sigma_3)W_9(t)\delta_6$$
$$\delta_4 = \sigma_4(1.0 - \sigma_4)W_{10}(t)\delta_6$$
$$\delta_5 = \sigma_5(1.0 - \sigma_5)W_{11}(t)\delta_6$$

- Adjust synaptic weights:

$$\Delta W_1(t) = \alpha\Delta W_1(t - \tfrac{1}{f}) + \eta\sigma_1\delta_3 \qquad \Delta W_5(t) = \alpha\Delta W_5(t - \tfrac{1}{f}) + \eta\sigma_2\delta_3$$
$$\Delta W_2(t) = \alpha\Delta W_2(t - \tfrac{1}{f}) + \eta\sigma_1\delta_4 \qquad \Delta W_6(t) = \alpha\Delta W_6(t - \tfrac{1}{f}) + \eta\sigma_2\delta_4$$
$$\Delta W_3(t) = \alpha\Delta W_3(t - \tfrac{1}{f}) + \eta\sigma_1\delta_5 \qquad \Delta W_7(t) = \alpha\Delta W_7(t - \tfrac{1}{f}) + \eta\sigma_2\delta_5$$
$$\Delta W_4(t) = \alpha\Delta W_4(t - \tfrac{1}{f}) + \eta\sigma_1\delta_6 \qquad \Delta W_8(t) = \alpha\Delta W_8(t - \tfrac{1}{f}) + \eta\sigma_2\delta_6$$

$$\Delta W_9(t) = \alpha\Delta W_9(t - \tfrac{1}{f}) + \eta\sigma_3\delta_6 \qquad \Delta W_{12}(t) = \alpha\Delta W_{12}(t - \tfrac{1}{f}) + \eta\delta_3$$
$$\Delta W_{10}(t) = \alpha\Delta W_{10}(t - \tfrac{1}{f}) + \eta\sigma_4\delta_6 \qquad \Delta W_{13}(t) = \alpha\Delta W_{13}(t - \tfrac{1}{f}) + \eta\delta_4$$
$$\Delta W_{11}(t) = \alpha\Delta W_{11}(t - \tfrac{1}{f}) + \eta\sigma_5\delta_6 \qquad \Delta W_{14}(t) = \alpha\Delta W_{14}(t - \tfrac{1}{f}) + \eta\delta_5$$

$$\Delta W_{15}(t) = \alpha\Delta W_{15}(t - \tfrac{1}{f}) + \eta\delta_6$$

$$W_n(t + \tfrac{1}{f}) = W_n(t) + \Delta W_n(t)$$

Figure 2.6.4: Summary of NMC equations for second order network and plant

Chapter 3: Experimental Results

For the first tests of the NMC algorithm, the plant transfer function $G(s) = 1/s^2$ was used. This was seen as the simplest nontrivial test case for the algorithm. For most of the experiments conducted with this system, the initial state was $\mathbf{x}_0^T = [0\ 0]$ and the desired final state was $\mathbf{x}_d^T = [1\ 0]$.

For the results described in this chapter, over seventy different experiments were performed assessing the impact of changing parameter values and operating environments on the behavior of the NMC algorithm. For each experiment, data was recorded about the state of the network and plant at 0.05 second intervals for the first five training phases and all fifty solo phases. This is an enormous amount of data, so clearly the following can present only a representative sample of the results. Even with this amount of data reduction, however, the reader may be somewhat overwhelmed by the proliferation of figures, tables, and equations in the sections which follow, so this introduction will conclude with a brief overview of the results to be presented.

Figure 3.0.1 shows the response of the system with an untaught network. It is clearly unstable, reflecting the total lack of knowledge about the plant at the startup of the algorithm. In the following, both the state weighting and model reference forms of the payoff function are examined. The parameters of these payoff functions and of the training algorithm itself are varied and the resulting variations in the control laws (if any) successfully devised by the network are assessed. Based upon the results of this search of the parameter space, a "canonical" form for the **M** matrix is determined and used as the reference value for subsequent experiments.

Section 3.1 describes the stability criteria which will be applied to the algorithm; these are somewhat more restrictive than just asymptotic stability of the plant, driven by a trained network, to the desired equilibrium. Section 3.2 details the analysis methods which will be employed and discusses some aspects of the operation of the algorithm. Section 3.3 examines in detail the state weighted form of the payoff function. The different parameters in the payoff function and the training algorithm itself are varied and the resulting impact on the control law developed (if any) by the network are evaluated. Section 3.4 performs a similar analysis for the model reference form of the payoff using several typical model trajectories. Finally, in Section 3.5, the robustness of the algorithm and trained network are tested in several different ways.

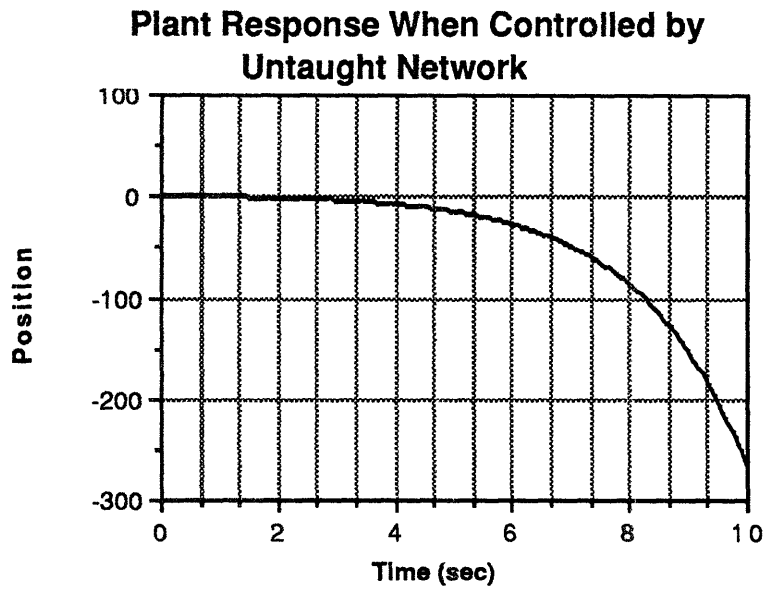

Figure 3.0.1: Response of double integrator plant when controlled by an untrained network.

3.1 Stability Conditions

In order to assess the performance of the network, there are four criteria which must be applied:

- whether the system (controller plus plant) is stable while training;
- whether the closed loop algorithm implemented by a trained network is asymptotically stable to the desired equilibrium point when the trainer is off-line;
- whether the training algorithm forces the network to develop a control law which is invariant as the number of training runs tends toward infinity; and,
- whether the responses seen while training and the responses obtained after each training run converge, in some sense, as the number of training runs increases.

Each of these criteria is a measure of the stability of the network. The first two are obvious measures of algorithmic stability, although note that the controller is permitted to "fail" a few times, i.e. the response of the system can exceed certain predetermined bounds as outlined in the previous chapter, before the first two stability measures are met. The third point is more subtle: even though the training and trained responses are stable, the network may be continually changing the control law it implements, for example commanding progressively higher bandwidth closed loop systems, after each training run. Such a network could not be claimed to be "stable" in any sense, since, even though after each training iteration the network implements a stabilizing control law, this control law never converges. This would require that the adaptation mechanism be arbitrarily "turned off" at some point to prevent further evolution of the control law, which is clearly undesirable since the algorithm would no longer be able to react to changes in the plant dynamics. It is thus important that the algorithm converge to a single control law for a time invariant plant; i.e. with everything else held constant, the network should eventually stop learning without the need to explicitly turn the trainer off. As will be shown, this is equivalent to requiring that the network synaptic weights converge to constant values after a finite number of runs.

The fourth point also rather subtle: it is necessary to ensure that there is not unfavorable interaction between the training algorithm and the control law being implemented. This would manifest as a gross discrepancy between the responses seen while training and the responses seen after training; for example, the closed loop response

may be nicely overdamped when the trainer is off, but very oscillatory when the trainer is turned on. This can be viewed as the network "fighting against the teacher", and is quite undesirable. Ideally, when the network is fully trained the responses seen in the training phase should be indistinguishable from those seen in the solo phase; this is really just another way of saying that the network stops learning in the steady state. This final measure of the network stability thus requires that the training and trained closed loop responses and control laws converge as the number of training runs increase.

3.2 Analysis Techniques

To understand the control laws being implemented by the network, it is necessary to analyze in some detail the topology of the synaptic connections. From inspection of Figure 2.13, the control law can be written as:

$$u = f_L(\mathbf{x}) + f_N(\mathbf{x}) + u_o \tag{3.1}$$

where $f_L(\mathbf{x})$ is the linear part of the control, $f_N(\mathbf{x})$ is the nonlinear part, and u_o is a constant bias. Since many of the problems to be examined will require zero steady state control to be applied when the plant is at the desired equilibrium, clearly we must have $-u_o = f_L(\mathbf{x}_d) + f_N(\mathbf{x}_d)$ in these cases. Referring again to Figure 2.13, it is obvious that, using the neuron and synaptic weight numbering scheme developed in Section 2.6 for a second order plant:

$$u_o = W_{15} \tag{3.2a}$$

$$f_L(\mathbf{x}) = W_4 x + W_8 \dot{x} \tag{3.2b}$$

$$f_N(\mathbf{x}) = W_9 \mathrm{sig}(q_3) + W_{10} \mathrm{sig}(q_4) + W_{11} \mathrm{sig}(q_5) \tag{3.2c}$$

where the q_i are the total inputs received by the hidden neurons, and the sigmoid function, sig(•), is given by equation (2.3). For the purposes of analysis only, we can approximate this function as piecewise linear, with:

$$\mathrm{sig}(q) = \begin{cases} 0 & \text{if } q < -3 \\ \frac{1}{6}(q+3) & \text{if } |q| \le 3 \\ 1 & \text{if } q > 3 \end{cases} \tag{3.3}$$

That this is is a reasonable approximation is seen from Figure 3.2.1; the largest deviation from the true sigmoid is about 0.06 units. With this simplification, the output of each neuron is essentially linear for weighted total inputs between -3 and 3; outside this region, the neuron is either saturated off, in which case it contributes nothing to the control signal, or saturated on, in which case it contributes W_n (n = 9, 10, or 11) to the control. Note that a neuron which is turned on may contribute either positive or negative control depending upon the sign of its associated W_n.

Recall that for each hidden neuron, the q_i are defined by:

$$\begin{aligned} q_3 &= W_1 x + W_5 \dot{x} + W_{12} \\ q_4 &= W_2 x + W_6 \dot{x} + W_{13} \\ q_5 &= W_3 x + W_7 \dot{x} + W_{14} \end{aligned} \tag{3.4}$$

Using equation (3.3), equation (3.4) can be re-arranged in terms of the plant state variables to determine the values of the state at which each neuron will turn on and off:

$$\begin{aligned} \text{ON:}\quad &\text{(Neuron 3):} && \dot{x} = -\frac{W_1}{W_5} x + \frac{(3 - W_{12})}{W_5} \\ &\text{(Neuron 4):} && \dot{x} = -\frac{W_2}{W_6} x + \frac{(3 - W_{13})}{W_6} \\ &\text{(Neuron 5):} && \dot{x} = -\frac{W_3}{W_7} x + \frac{(3 - W_{14})}{W_7} \end{aligned} \tag{3.5}$$

$$\begin{aligned} \text{OFF:}\quad &\text{(Neuron 3):} && \dot{x} = -\frac{W_1}{W_5} x - \frac{(3 + W_{12})}{W_5} \\ &\text{(Neuron 4):} && \dot{x} = -\frac{W_2}{W_6} x - \frac{(3 + W_{13})}{W_6} \\ &\text{(Neuron 5):} && \dot{x} = -\frac{W_3}{W_7} x - \frac{(3 + W_{14})}{W_7} \end{aligned}$$

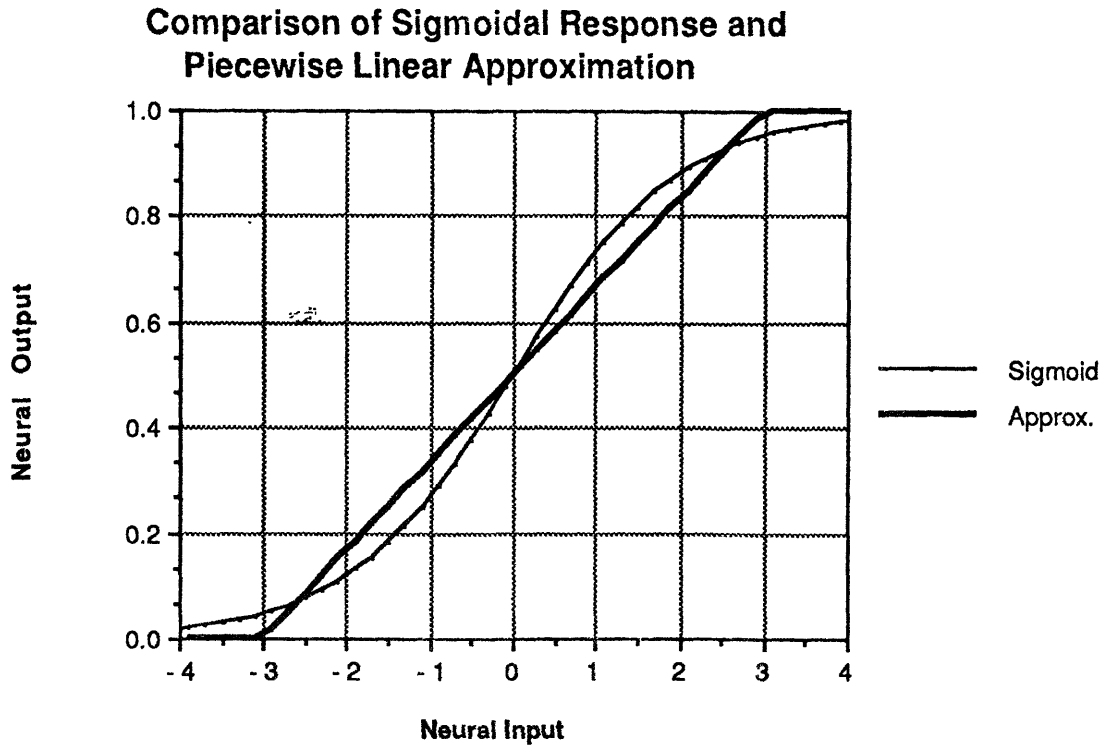

Figure 3.2.1: Comparison of true sigmoid with approximation (3.3)

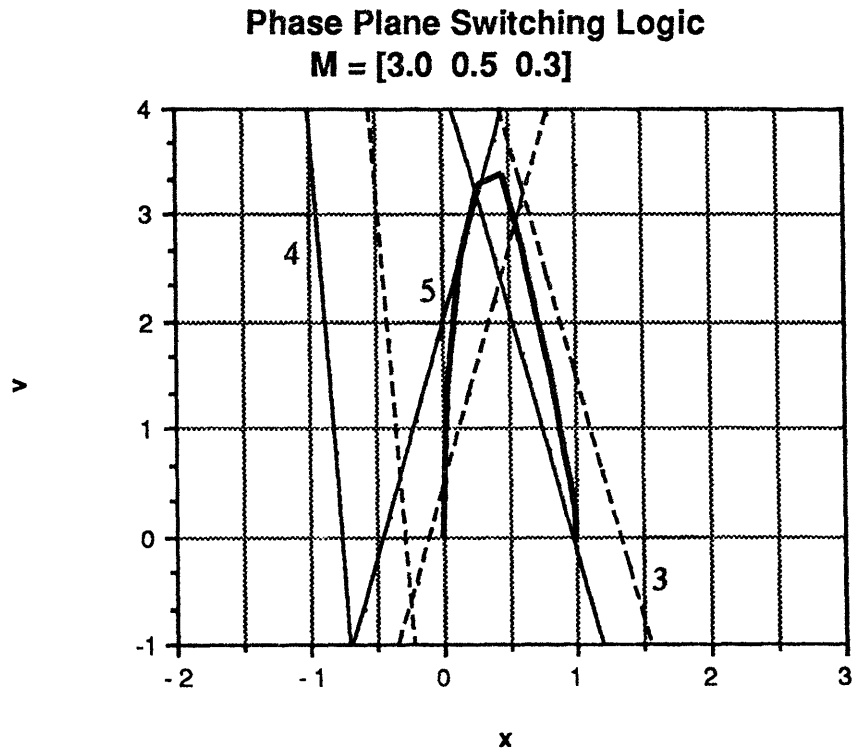

Figure 3.2.2: Typical switching curves developed by a network

The adaptive weights on each of the hidden neurons thus define *switching lines* on the state space which govern the behavior of the nonlinear half of the control law implemented by the network.

Figure 3.2.2 shows a typical example of this switching logic for a second order plant. The dark black border with a number beside it indicates the line at which the indicated neuron switches on; for all points in the state space to the left of this line the neuron will be fully on. The thinner, dashed borders indicate the lines where each neuron switches off; it is clear from equations (3.5) that each off switching line will always parallel the corresponding on switching line. The area between corresponding on and off switching lines indicates the region of state space where each neuron's output is roughly linear, according to the above equation (3.3). By plotting the trajectory (the heavy, bold line starting at the origin) of the simulated response through the state space and noting where it intersects the switching lines for each neuron, it is possible to gain valuable insight into both the inner workings of the network and the control law being employed.

From the above discussion, the control law being implemented by a trained network can be seen as piecewise linear across the state space with discontinuities in slope occurring at the switching lines. For two dimensional plants, it is possible to visualize the control law by plotting, in three dimensions, the control versus the two state variables. For a purely linear control law, this plot will be a smooth two dimensional plane with a continuous slope, as shown in Figure 3.2.3; the steeper this slope, the larger the control action and hence bandwidth of the closed loop system. The control law being implemented by the neuromorphic controller, however, will resemble this linear controller only in regions far from the switching lines. In the vicinity of these nonlinear boundaries the surface will wrinkle, creating steep slopes and plateaus in the otherwise smoothly sloping surface, as shown in Figure 3.2.4. Thus, when the system is operating in the saturation region (either on or off) of all the hidden neurons, the plateaus will have the same slope as the constant control surface which would result from just the linear half of the control law, although offset in absolute terms. When the system operates in the linear regions of the hidden neurons, however, the trajectories move along the "steep slopes" which connect the plateaus and hence the system exhibits higher bandwidth behavior.

Due to the crudeness and lack of detail, these three dimensional figures are included for illustration purposes only, and will not be used for numerical analysis in the following. The switching line plots of Figure 3.2.2, however, will be used extensively.

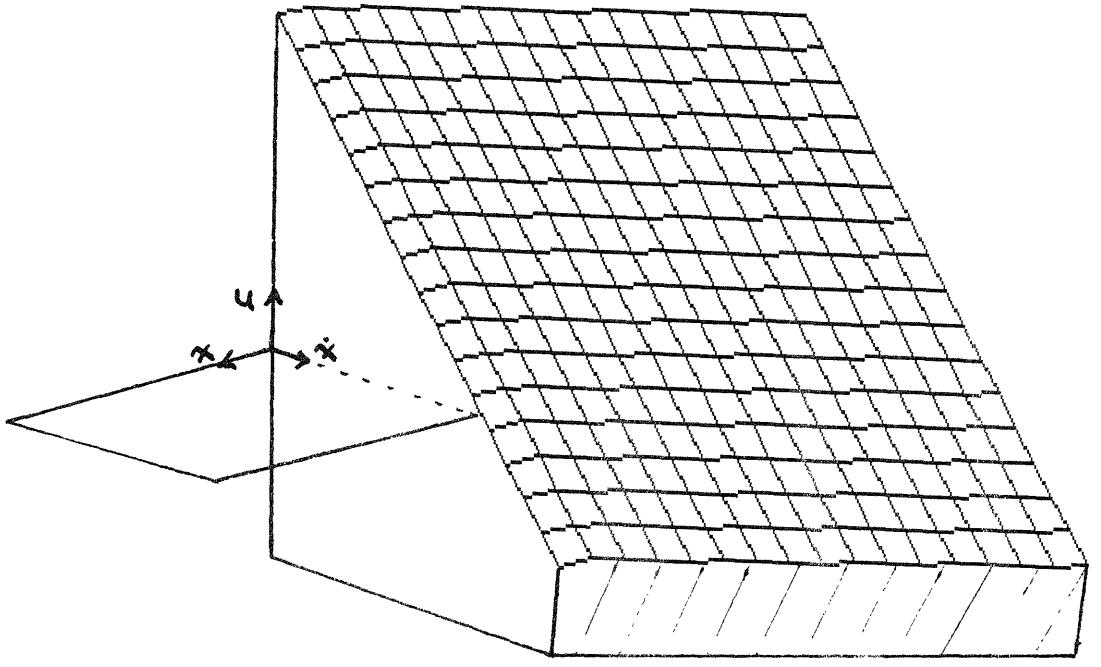

Figure 3.2.3: Control "surface" for a linear two dimensional plant

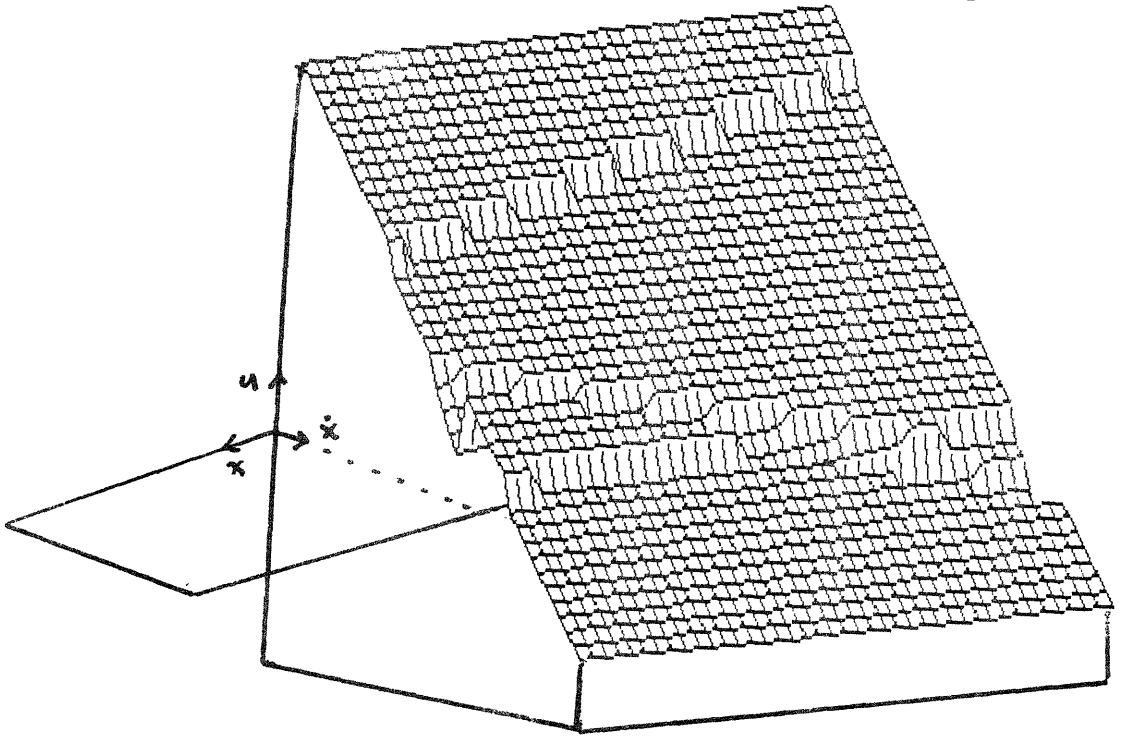

Figure 3.2.4: Control "surface" developed by a typical network, corresponding to the switching lines shown in Figure 3.2.3.

From this analysis one can see that each hidden neuron makes a contribution to the nonlinear half of the network's control law. This contribution has four degrees of freedom corresponding to the four adaptive weights associated with each. Two of these degrees of freedom specify the orientation (slope and intercept) of the switching lines in the phase space. A third degree of freedom specifies the *width* of the linear region between the on and off switching lines. Finally, the fourth degree of freedom specifies the magnitude and sign of the maximum contribution of each neuron. The control law implemented by this network thus has a total of fifteen degrees of freedom (four each for the three hidden neurons, and one each for the position, velocity, and control bias neurons) corresponding to the fifteen synaptic weights. It is these degrees of freedom which are tuned by the interaction of the network with the trainer and the plant dynamics.

It is possible for the NMC algorithm to use this freedom to design switching lines such that, as the state evolves, the hidden neurons are always operating in their linear region or in saturation. In this case, the controller implemented by the network will be referred to as "linear", even though globally, of course, the controller is far from linear. The control law will be referred to as "nonlinear" only in those cases when the states seen in the simulations actually cross one or more of the switching lines. The individual contributions of each neuron to the control law will still be referred to as "linear" or "nonlinear" depending whether they come from the input (linear) neurons or the hidden (sigmoidal) neurons.

It has already been noted that the back propagation algorithm defines a method of changing synaptic weights such that the error signal at the output layer is minimized, ideally driven to zero. If this result holds true for the NMC algorithm (which, recall, is not using pure back propagation techniques), $\delta(t)$ will be driven to zero in finite time and the network will thus stop learning. Hence, at least one of the above stability considerations cited above can be met by ensuring that $\delta(t)$ is zero (or indistinguishably close thereto) *for all time steps* in a perfectly trained network. Since $\delta(t)$ is computed anew at each point in time based only upon the *instantaneous* current values of x, $\dot{x}$, and u, if the algorithm works as anticipated the network will implement a mapping which forces $\delta(t)$ to zero *pointwise in time*. The control law thus generated will arise not as a global optimization of some performance metric, but rather as a minimization of the value of $\delta(t)$ at each time step. The form of the payoff signal, at least for the state weighted payoff function, hence implicitly defines an "ideal" tradeoff between the values of x, $\dot{x}$, and u at each point in time.

In this sense, the algorithm has no sense of "temporal" optimality; it does not look at the overall behavior of the system, only how far it is deviating from an implicit ideal at

each time step. This potentially places a limitation on the kinds of systems which can be controlled by the NMC. In fact, an argument can be made that the payoff signal implicitly specifies a control law to the network, albeit in a quite roundabout fashion, since $\delta(t) \equiv 0.0\ \forall t$ is an implicit equation for u(t). (Of course, even were this argument correct, it would still be a noteworthy result that the NMC architecture has successfully untangled this implicit control law and devised a pattern of synaptic weights which implements it!) Two observations, however, refute this: first, the algorithm develops networks which implement stabilizing control laws even in the absence of a control weighting term in the payoff, thus making the above rearrangement for u(t) impossible; and second, in arriving at its steady state control law, the algorithm makes certain tradeoffs and adjustments which would be impossible to predict based solely upon inspection of the payoff function itself--these tradeoffs arise only through interaction with the plant dynamics during training. Experimental evidence for each of these assertions will be presented below.

It is thus useful to keep in mind while reading reading the following results that the NMC is attempting to drive $\delta(t)$ to zero at each instant in time, subject to the (unknown) relations between the state variables and controls as they appear in the payoff function. In general, the actual control law implemented by the network will arise through a complicated interaction of the payoff signal, plant dynamics, and range of plant states visited while learning.

3.3 State Weighting Technique

Recall that the payoff function for a second order, single-input, single-output system can be expressed as:

$$\delta(t) = M_x(x_d - x(t)) + M_{\dot{x}}(\dot{x}_d - \dot{x}(t)) - M_u \mathrm{sgn}(u)\left|\frac{u}{u_{ult}}\right|^n \qquad (3.7)$$

In the following the (scalar) weighting parameters will be referred to as $\mathbf{m}^T = [M_x \quad M_{\dot{x}} \quad M_u]$. u_{ult} was set to 16. This is somewhat arbitrary, but serves as a reference, and is actually a physically meaningful parameter for the envisioned hardware application of NMC, which will be explained in the final chapter.

try to bring the payoff signal, δ, to zero, or a minimum, as quickly as possible. This is similar to the LQ problem, where one attempts to find a control law which minimizes the cost function:

$$J = \int_{-\infty}^{\infty} \{\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{u}^T\mathbf{R}\mathbf{u}\}\, dt \tag{3.8}$$

or, for the single input and output case:

$$J = \int_{-\infty}^{\infty} \{\mathbf{x}^T\mathbf{Q}\mathbf{x} + \rho u\}\, dt \tag{3.9}$$

Comparing these two equations it is possible to see a connection between M_x, $M_{\dot{x}}$, and the diagonal elements of the **Q** matrix, although in the NMC algorithm these constants weight the states themselves instead of the squares of the states. There is actually a very important reason for this difference, and this will be discussed at the end of the next chapter. It is thus possible to draw on intuition gained from the multitude of research which has been conducted on the LQ problem. One would expect to the NMC algorithm develop a control law which yields an overdamped response for values of the ratio of M_x to $M_{\dot{x}}$ greater than or approaching one, and an underdamped response for values of this ratio significantly less than one. Similarly, the ratio $\frac{M_u}{u_{ult}^4}$ is analogous to the control weighting term ρ in equation (3.9), although control magnitudes are more severely penalized in NMC. One would thus expect to see very large bandwidth controllers developed by NMC as $M_u \rightarrow 0$, analogous to the "cheap" control LQ problem, and much lower bandwidth controllers as M_u increases.

3.3.1 Position Weighting Only

Figure 3.3.1 shows the response of the system during its first training runs, and Figure 3.3.2 shows the response with the trainer off for $\mathbf{m}^T$ = [1.0 0.0 0.0], and n = 4. Notice that the NMC has devised a stabilizing control signal during its first training

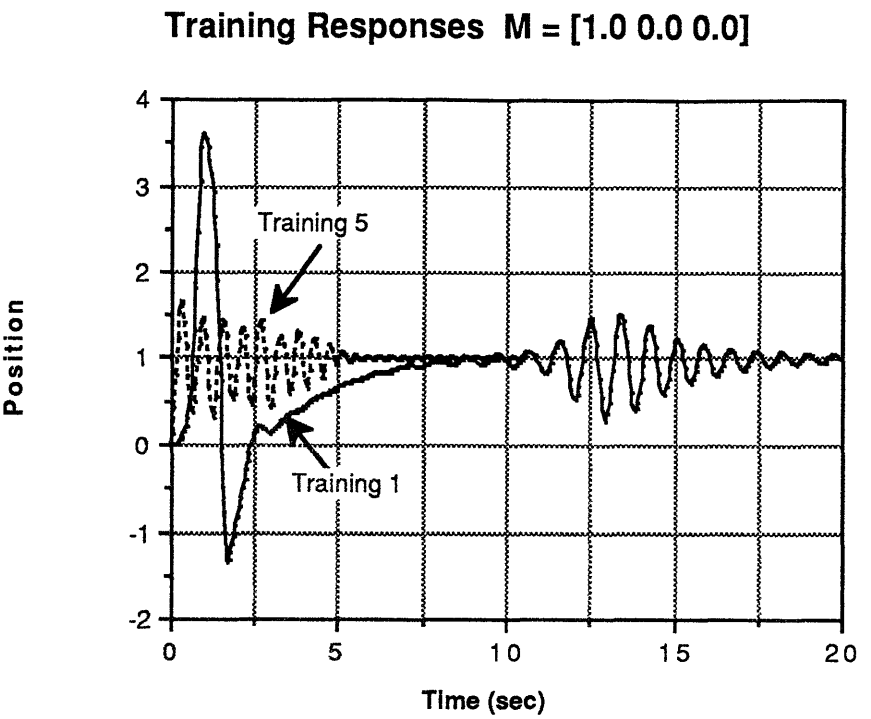

Figure 3.3.1: Response of system during first training phases: $\mathbf{m}^T = [1.0 \ 0.0 \ 0.0]$

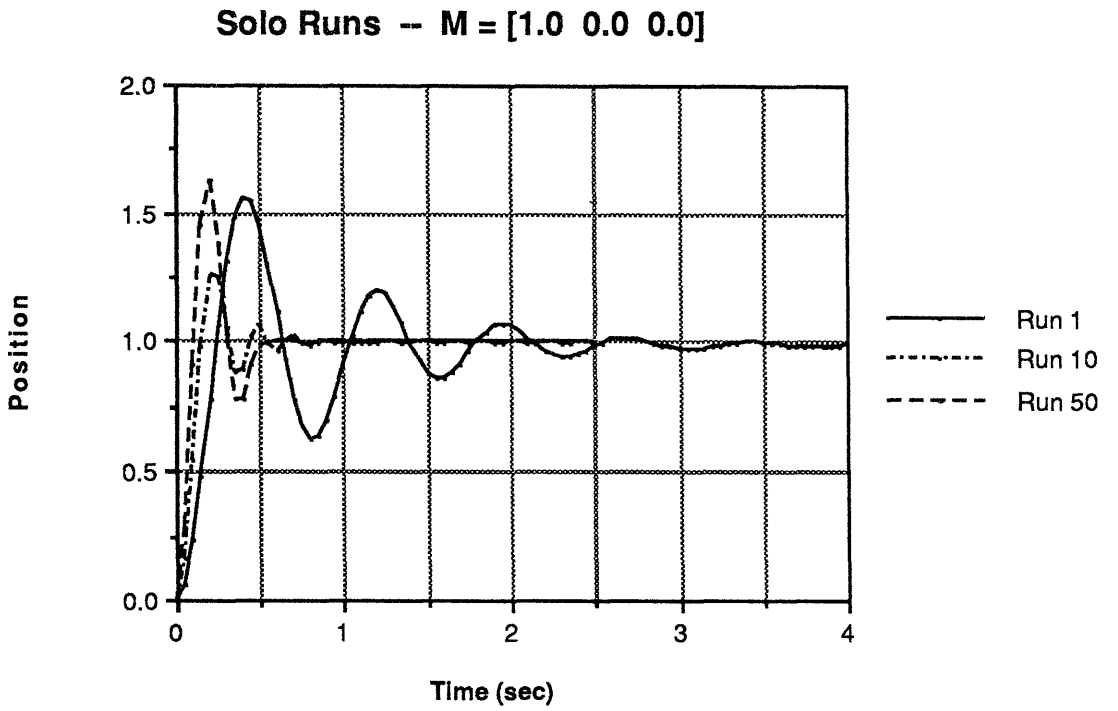

Figure 3.3.2: Response of system during first solo phases: $\mathbf{m}^T = [1.0 \ 0.0 \ 0.0]$

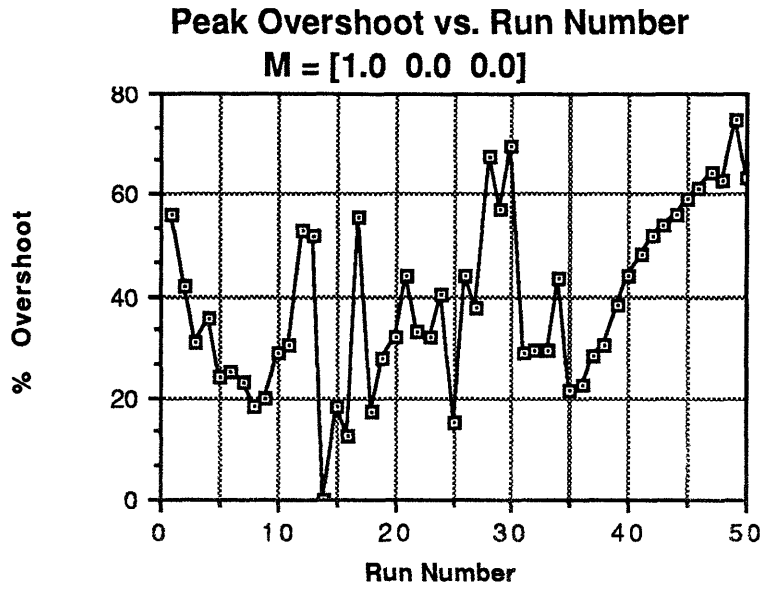

Figure 3.3.3: Maximum overshoots of closed loop responses

run, notice further that the network never "failed"; i.e. its response while training stayed within the allowable bounds. Recall that the XOR problem, for example, required about 500 iterations of the training set to converge on a set of network weights which solved this problem. Since each training run of NMC involves (nominally) 400 weight update cycles, it is reasonable to expect at least a partial solution to the control problem after only one training run. It thus appears that despite the reservations expressed in the previous chapter about infinite training sets, a stabilizing control signal has been constructed, *for at least the limited part of the state space the network does experience while training*. The fact that the network was started from the same point in state space at each iteration of the training process means that the system has experienced only a small fraction of the total possible state space. This observation represents an important limitation in the way these experiments were performed, and will be addressed at greater length in Section 4.1.5 and in the Conclusions.

For these initial payoff weightings there appears to be no correlation as to the quality of the solutions obtained as a function of the number of training runs. In fact, the solutions are not very good as they tend to overshoot quite a bit. Even though the amount of overshoot tends to change after each training run, there is no obvious trend to these changes, as Figure 3.3.3 shows. This is clearly a case where the control law does not converge; as Figure 3.3.4 shows, the shape of the control law is similar from run to run, but the maximum amount of control commanded (which occurs at t = 0) increases nearly

linearly with the number of runs, as illustrated in Figure 3.3.5. Given an "infinite" number of training runs training runs, the NMC would begin to command huge amounts of control and hence create effectively infinite bandwidth closed loop controllers. For a physical system, this is clearly undesirable since high frequency unmodelled effects (e.g. structural modes, sensor and actuation delays, etc) would render the training model invalid. The algorithm with $\mathbf{m}^T = [1.0\ 0.0\ 0.0]$, or more generally $\mathbf{m}^T = [M_x\ 0.0\ 0.0]$, is thus *unstable* from the definition given above.

Note, however, that this "instability" is not unexpected in light of the comparison between LQ and NMC discussed above. For this payoff function, with no control weighting, one would expect to see infinite bandwidth controllers develop; there is nothing in the penalty function which prevents it.

3.3.2 Position and Velocity Weighting

Figure 3.3.6 shows the responses generated by the network on the fiftieth training run for a payoff function with nonzero velocity deviation weightings. notice that for even the smallest velocity weightings, the trained response is much more damped than with no velocity weighting. For increasing values of $M_{\dot{x}}$ the response becomes overdamped instead of oscillatory. However, as Figure 3.3.7 shows, the amount of control commanded by the network is still increasing linearly in time; NMC is again trying to command an infinite bandwidth controller. The algorithm with $\mathbf{m}^T = [M_x\ M_{\dot{x}}\ 0.0]$ is thus also *unstable*.

Once again, these results make sense in light of the comparison with the LQ problem. Decreasing the ratio of position to velocity weightings leads to responses which exhibit more and more damping, and, as above, the absence of any control weighting still leads to controllers with unlimited bandwidth.

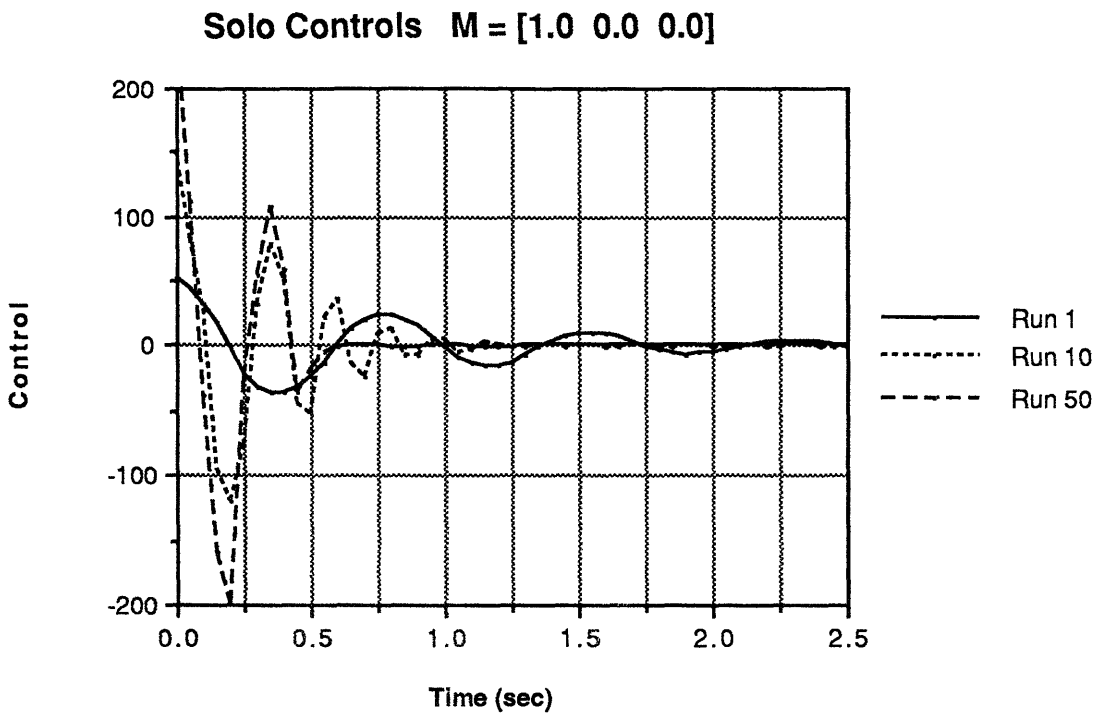

Figure 3.3.4: Control signals during solo phases: $\mathbf{m}^T = [1.0\ 0.0\ 0.0]$

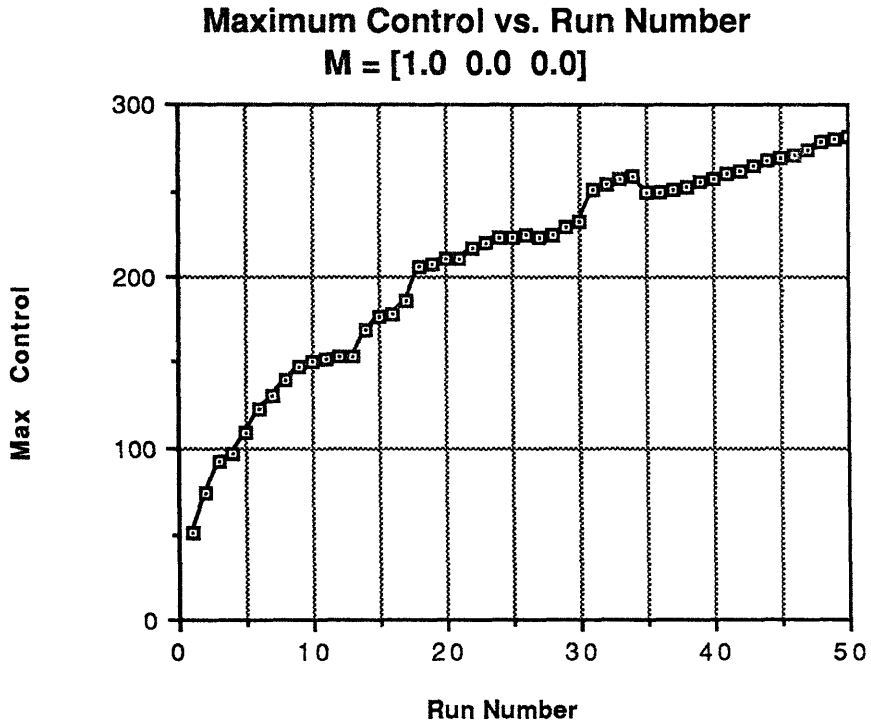

Figure 3.3.5: Maximum control usage during solo phases: $\mathbf{m}^T = [1.0\ 0.0\ 0.0]$

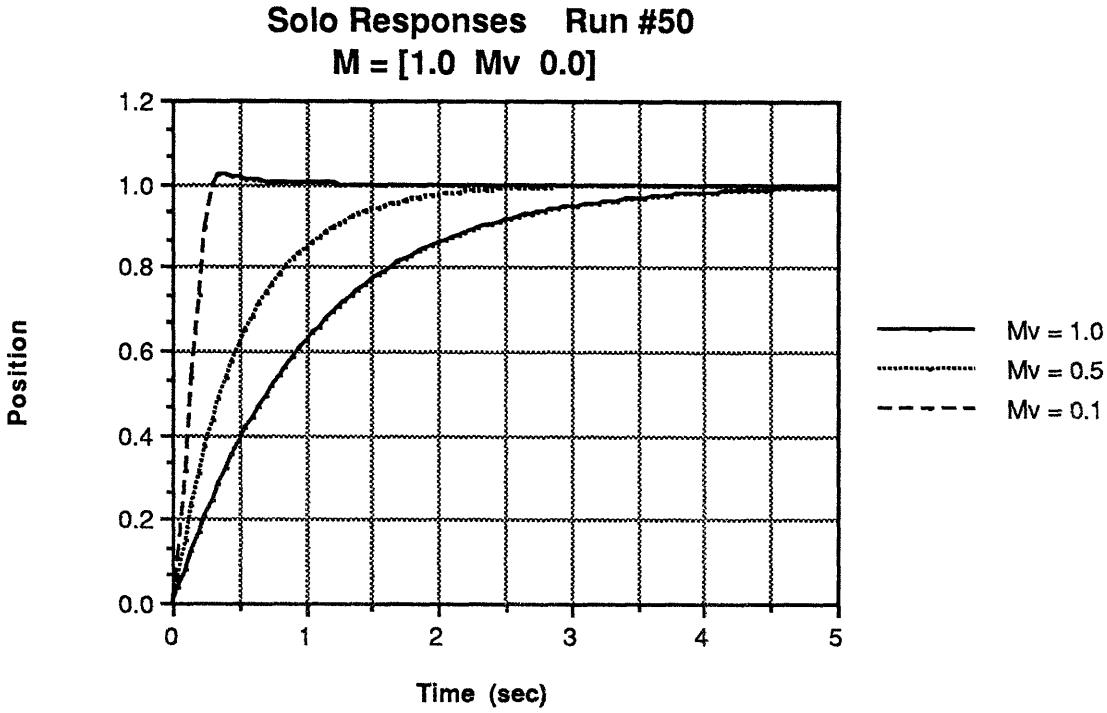

Figure 3.3.6: Plant responses during solo phases: $\mathbf{m}^T = [1.0\ M_{\dot{x}}\ 0.0]$

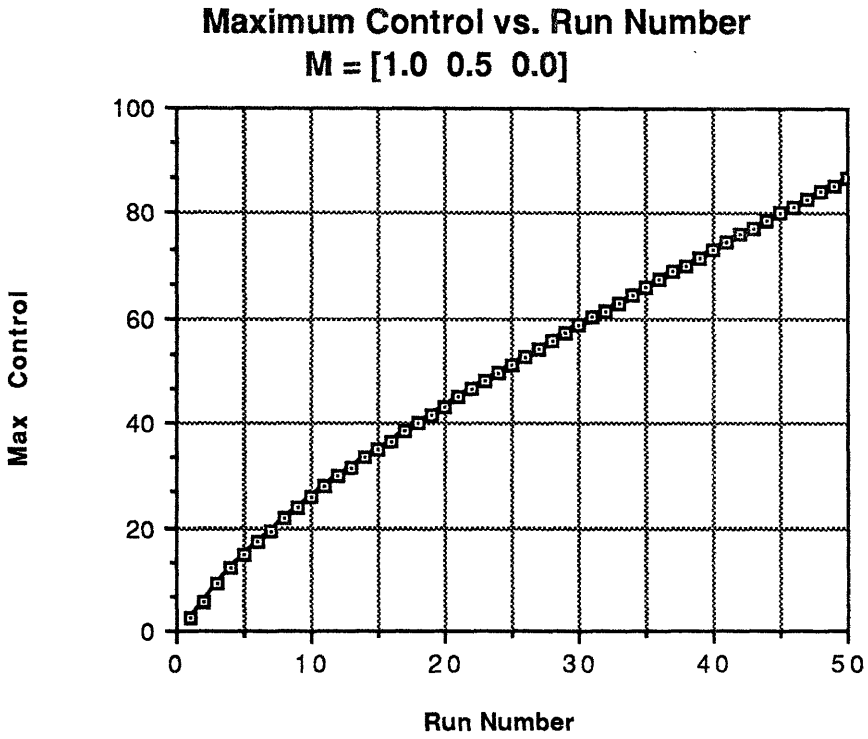

Figure 3.3.7: Maximum control usage during solo phases: $\mathbf{m}^T = [1.0\ 0.5\ 0.0]$

3.3.3 Position, Velocity, and Control Weighting

Figure 3.3.8 shows the responses generated after the fiftieth training run when all three elements of the weighting matrix are nonzero and Figure 3.3.9 shows the control signals used to generate these responses. Again the responses are overdamped, but as Figure 3.3.10 dramatically illustrates, the amount of maximum control used to generate the responses is constrained; the NMC is now converging to a single control law for the plant. To illustrate this, Figure 3.3.11 shows the solo responses generated by the network when trained with the payoff function weights $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$ during several different solo phases. Note that by the tenth run the closed loop response is almost identical to the fiftieth response shown in the previous figure. This is yet another indication that the algorithm is converging to a single control law. Finally, Figures 3.3.12 (a)-(c) compare the behavior of the system during the training and solo phases of several different runs. Notice that while initially the network does tend to "fight the teacher" on the first training run, by the twenty-fifth run the training and solo responses are almost indistinguishable.

To ensure that this convergence is absolute, a simulation with $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.7]$ was extended to 1000 training runs; the maximum control used after each run is plotted in Figure 3.3.13. Since it is (remotely) possible that the maximum control could be bounded while the weights of the network grow unboundedly, Figure 3.3.14 displays the values of the synaptic weightings of this network as a function of the run number. Clearly these, too, converge to finite values. The network has, as desired, essentially stopped, or at least greatly slowed, learning after only a few tens of runs, and thus has converged to a steady state network configuration which implements its control law. Notice that fifty runs was not quite enough for the synaptic weights and hence controls to converge to their final values, in fact these values do not totally stabilize for nearly 250 runs. However, the discrepancy between the values of u_{max} on the fiftieth run and those on the 250th is less than 8%, although the network synaptic weights vary considerably more.

Thus, the state weighting payoff function with a full weighting vector satisfies, for this plant, all of the conditions discussed in Section 3.1 for algorithmic stability. The training and solo phases are asymptotically stable to the desired equilibrium and the responses seen in each phase become indistinguishable after about twenty-five runs. The control law developed is essentially invariant after only fifty runs, and remains so for at least 1000 runs.

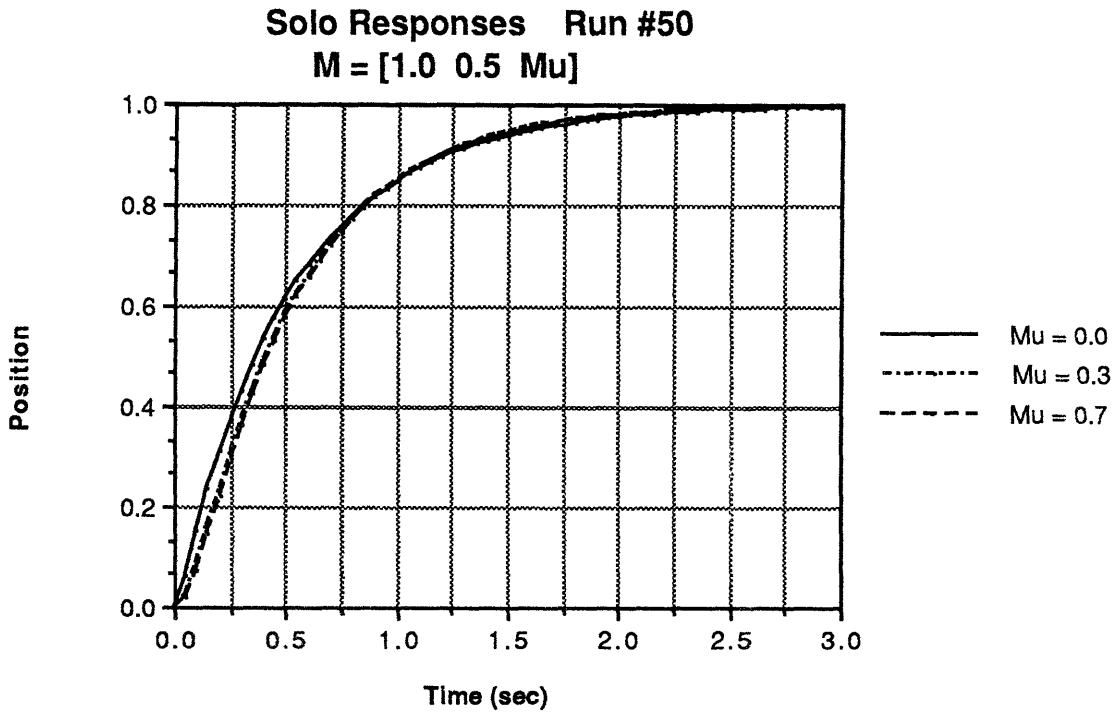

Figure 3.3.8: Solo responses, run 50: $\mathbf{m}^T = [1.0\ 0.5\ M_u]$

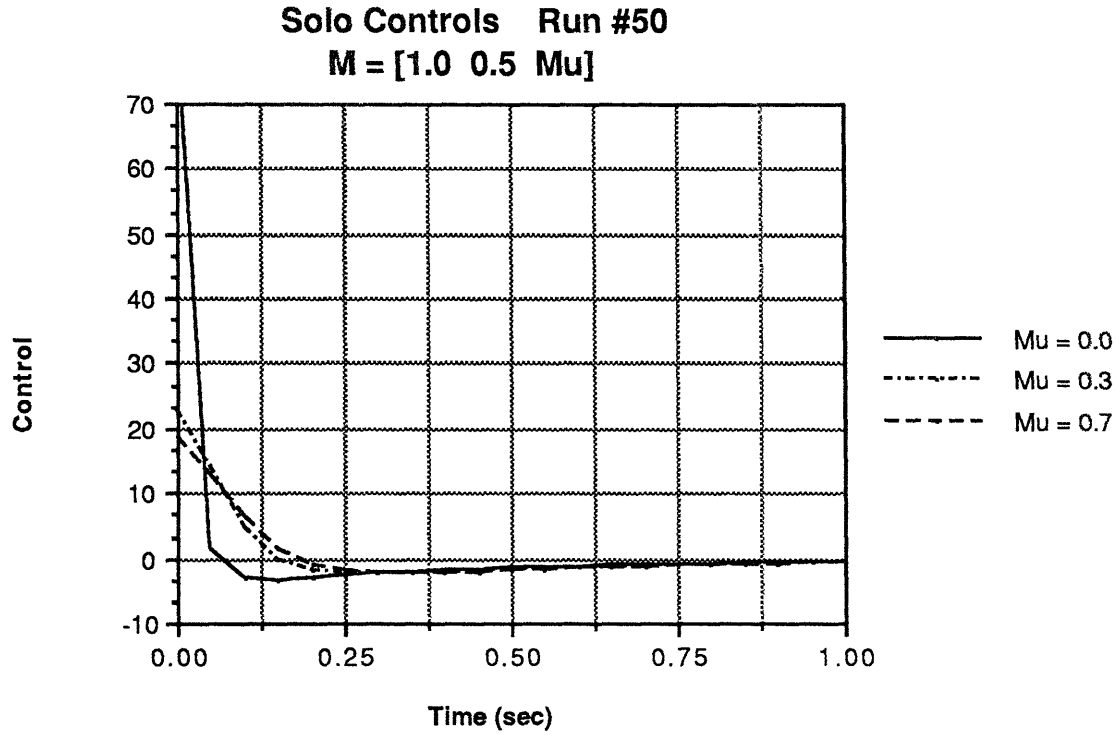

Figure 3.3.9: Control signals generated during solo run 50: $\mathbf{m}^T = [1.0\ 0.5\ M_u]$

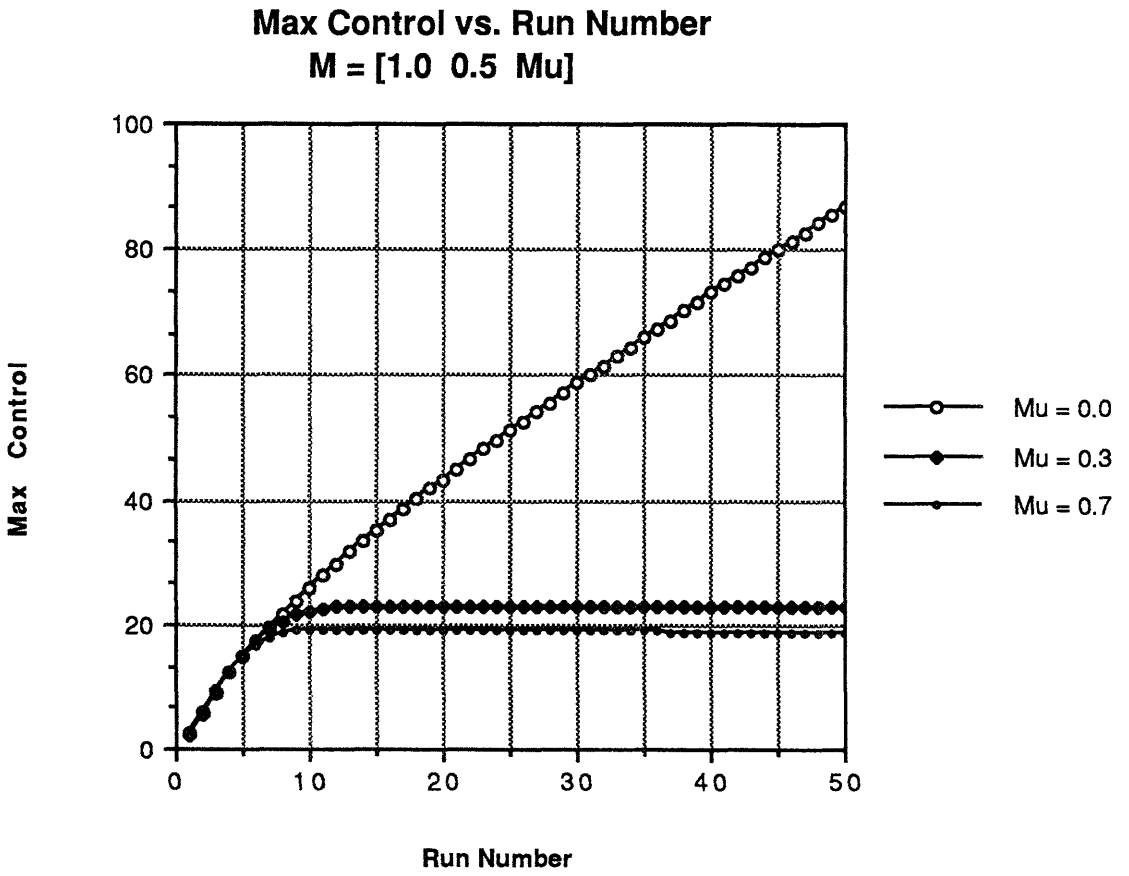

Figure 3.3.10: Maximum control used with different control weightings

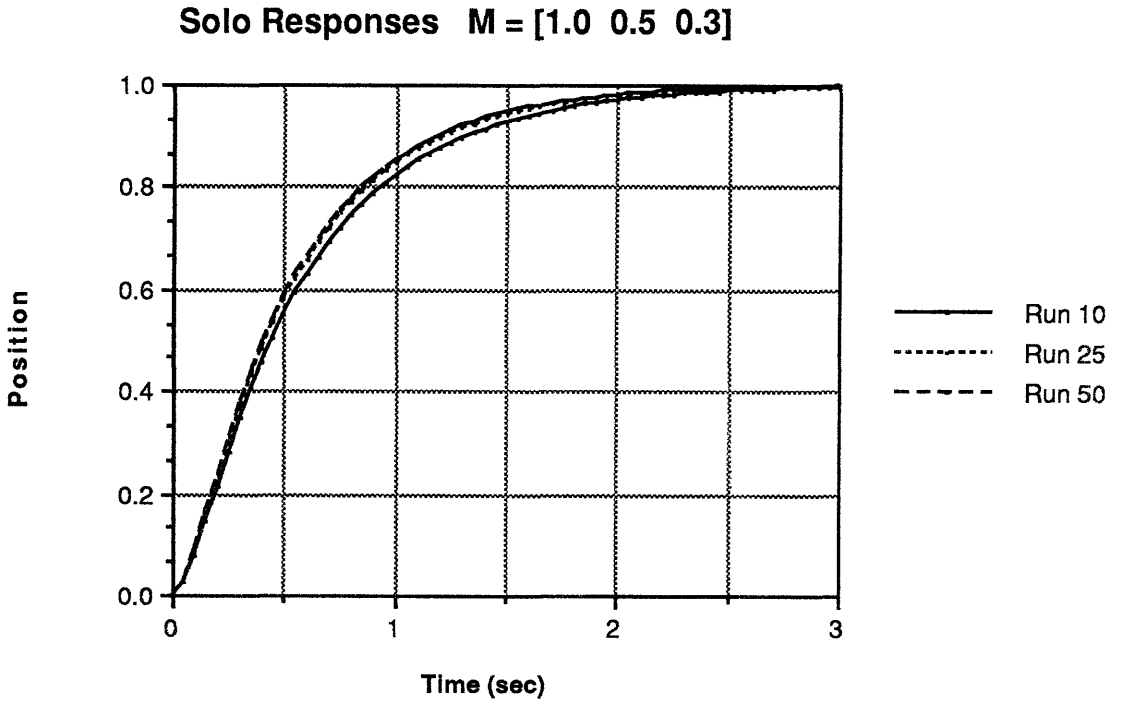

Figure 3.3.11: Comparison of different solo responses: $\mathbf{m}^T$ = [1.0 0.5 0.3]

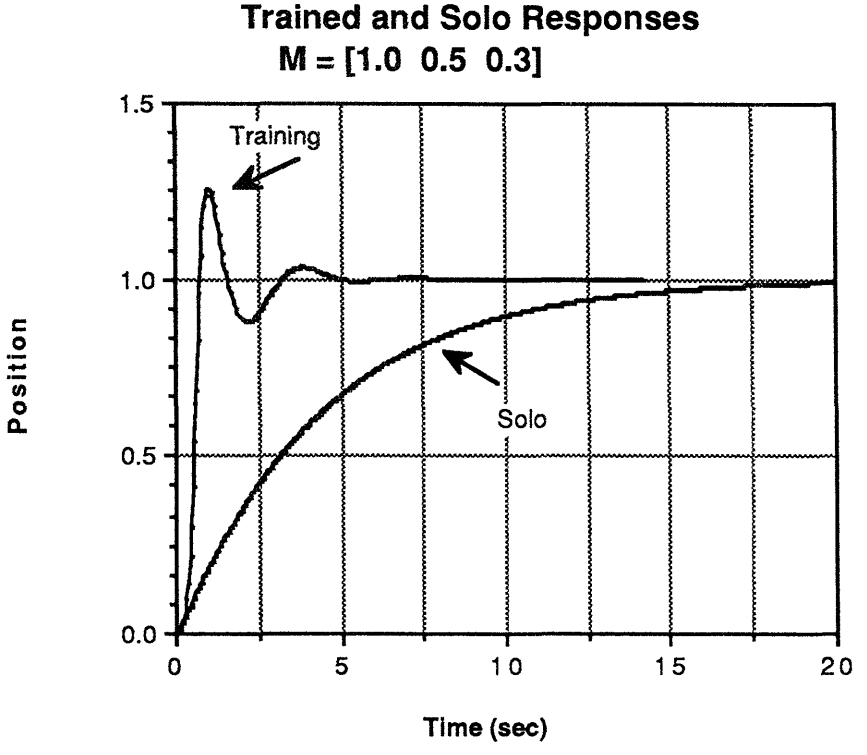

Figure 3.3.12(a): Comparison of training and solo responses, run 1

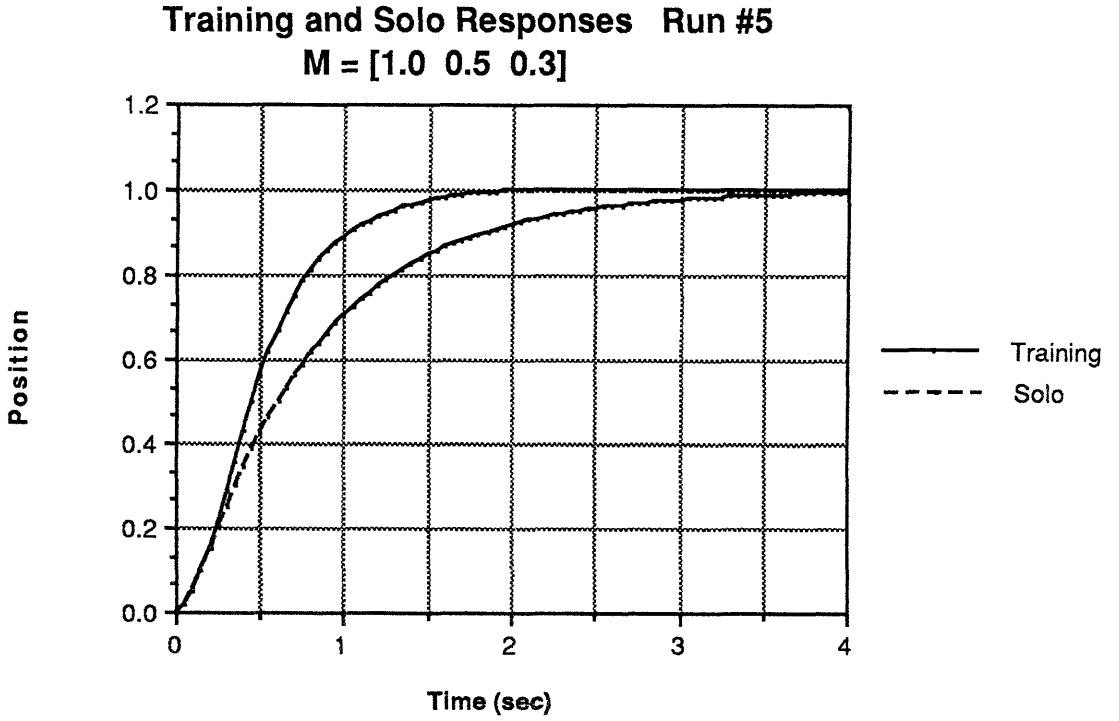

Figure 3.3.12(b): Comparison of training and solo responses, run 5

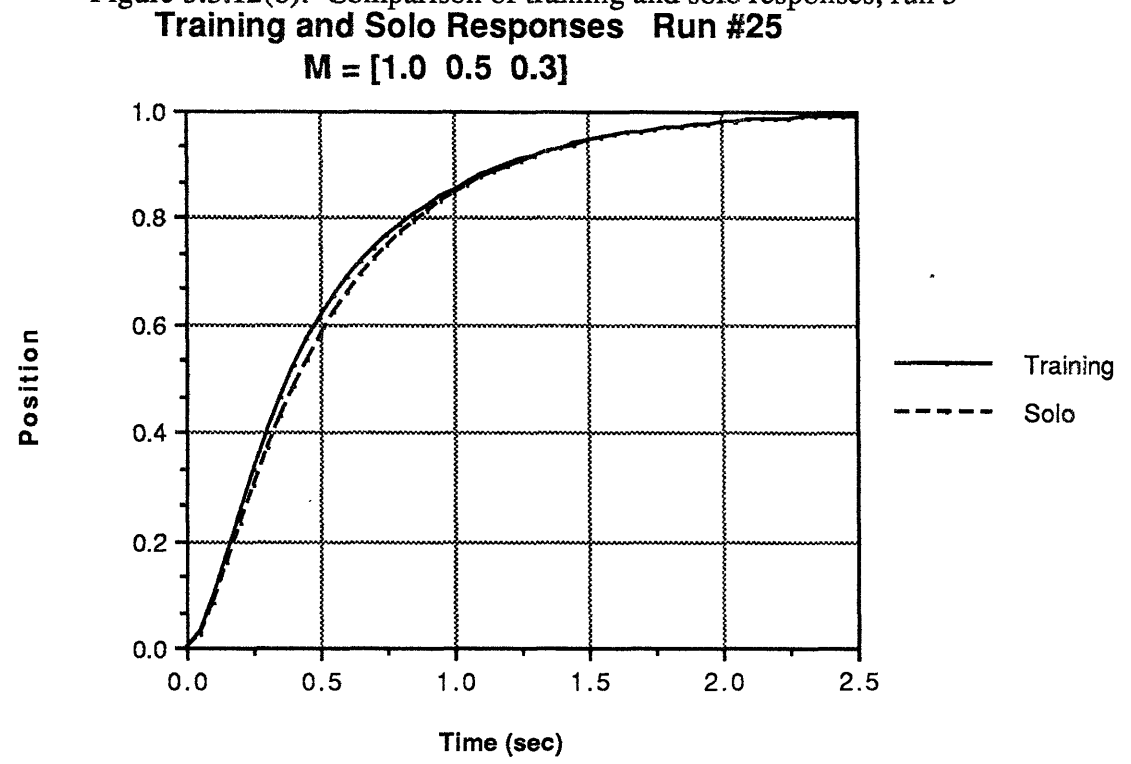

Figure 3.3.12(c): Comparison of training and solo responses, run 25

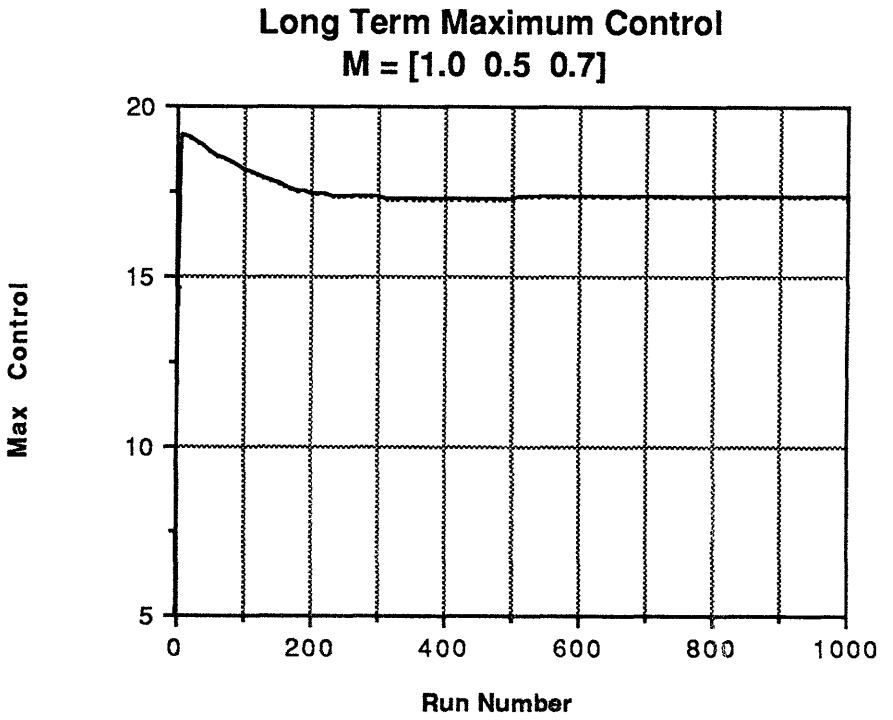

Figure 3.3.13: Maximum control used during solo phases: $\mathbf{m}^T = [1.0\ \ 0.5\ \ 0.7]$

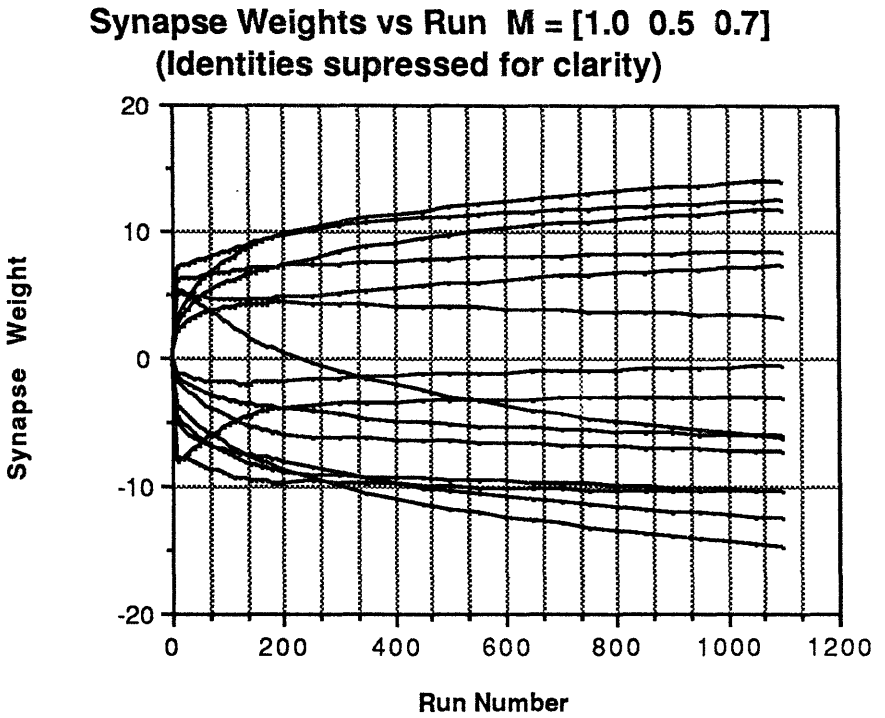

Figure 3.3.14: Synaptic weights after each run: $\mathbf{m}^T = [1.0\ \ 0.5\ \ 0.7]$

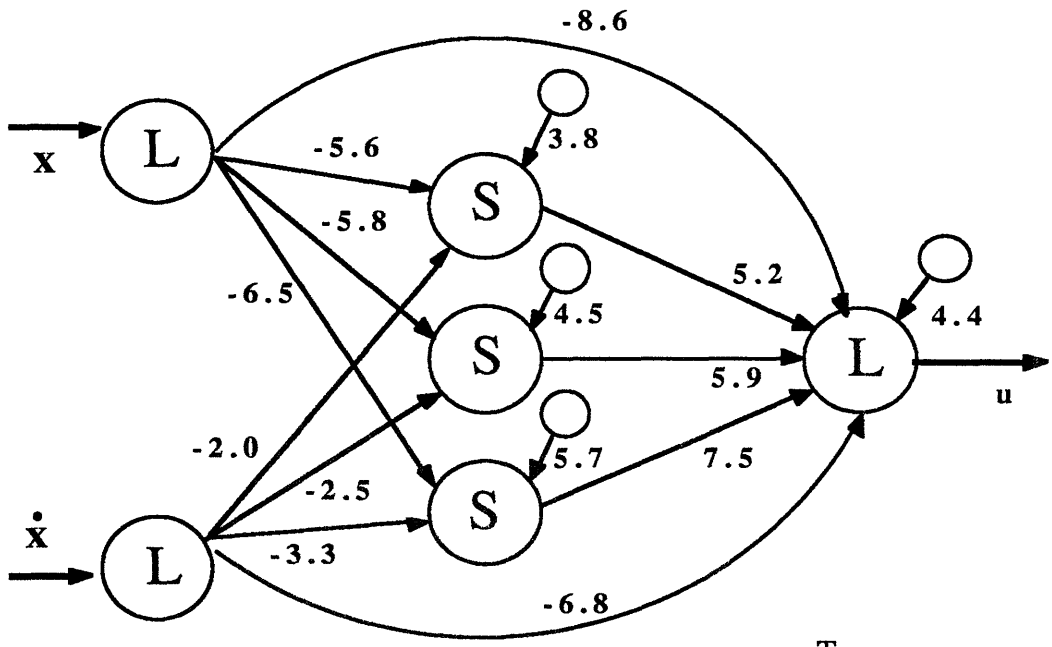

Figure 3.3.15: Network configuration after 50 iterations: $\mathbf{m}^T = [1.0\ \ 0.5\ \ 0.3]$

Based upon the above parametric analysis of the variables in the payoff function, the weight vector $\mathbf{m}^T = [1.0\ \ 0.5\ \ 0.3]$ was chosen as the "canonical" weighting, and the responses generated with this weight vector (Figures 3.3.8 - 3.3.12) will be used as the baseline against which other experimental results will be compared.

3.3.4 Solution Analysis for the Canonical Weighting

Having shown that the NMC algorithm using a state weighted payoff function is stable, in all the senses discussed in Section 3.1, for intuitively reasonable values of the weighting parameters, it is necessary to examine exactly how the network is implementing its control law. Figure 3.3.15 shows the configuration of the network after the fiftieth iteration of the algorithm for the canonical weighting examined above. From this diagram and with reference to equation (3.2), the control law can be written as:

$$u = 4.4 - 8.6x - 6.8\dot{x} + 5.2\sigma_2 + 5.9\sigma_3 + 7.5\sigma_4 \qquad (3.11)$$

where,

$$\begin{aligned} \sigma_2 &= \text{sig}(3.8 - 5.6x - 2.0\dot{x}) \\ \sigma_3 &= \text{sig}(4.5 - 5.8x - 2.5\dot{x}) \\ \sigma_4 &= \text{sig}(5.7 - 6.5x - 3.3\dot{x}) \end{aligned} \qquad (3.12)$$

Since the equilibrium point is $\mathbf{x}_d^T = [1.0\ \ 0.0]$, and since, for this system, the steady state control must be zero to maintain this equilibrium:

$$0 = -4.2 + 5.2\text{sig}(-1.8) + 5.9\text{sig}(-1.3) + 7.5\text{sig}(-0.8) \qquad (3.13)$$

which is indeed an equality (there is a slight roundoff error since the constants have been recorded here to only one decimal place accuracy). Thus the *entire network* contributes to the steady state control! This is dramatically different than the solution which would have been expected using the hypothesized "simplest case" controller discussed in Section 2.6.

Figure 3.3.16 shows how the magnitude of the payoff function varies during the first and twenty-fifth training runs. Notice that the absolute magnitude is lower at every point in time in the later training sequence. These curves support the above analysis which predicted the NMC algorithm would attempt to design synaptic weights so as to drive the payoff function to zero for all time. Many more examples in support of this observation will be given below.

Figure 3.3.17 shows the results of fitting a linear control law to the data obtained from this experiment. The fit is very close, so that to a good approximation the control law in equation (3.10) is given by:

$$u = 24.6(1.0 - x) - 13.4\dot{x} \qquad (3.13)$$

When this control law is used on the plant, the response shown in Figure 3.3.18 results. Note that this is very close to the actual response observed, which strengthens the assertion that, despite the nonlinearities in the control law, the NMC algorithm has developed a linear controller.

The phase space switching lines shown in Figure 3.3.19 confirm this analysis. Except for values of the state relatively far from the equilibrium point, the hidden neurons are all operating in their linear region. Recall from the discussion in Section 3.2 that, when the hidden neurons operate linearly, the state is moving along the steep slopes in the control surface which connect the saturation region "plateaus". Because of the overlap of the switching lines for this system, there is only one such, very steep, "wrinkle" in the control surface and the control generated by the trajectory of the system remains on this wrinkle for the duration of the simulation. Figure 3.3.20 gives a very crude picture of this situation.

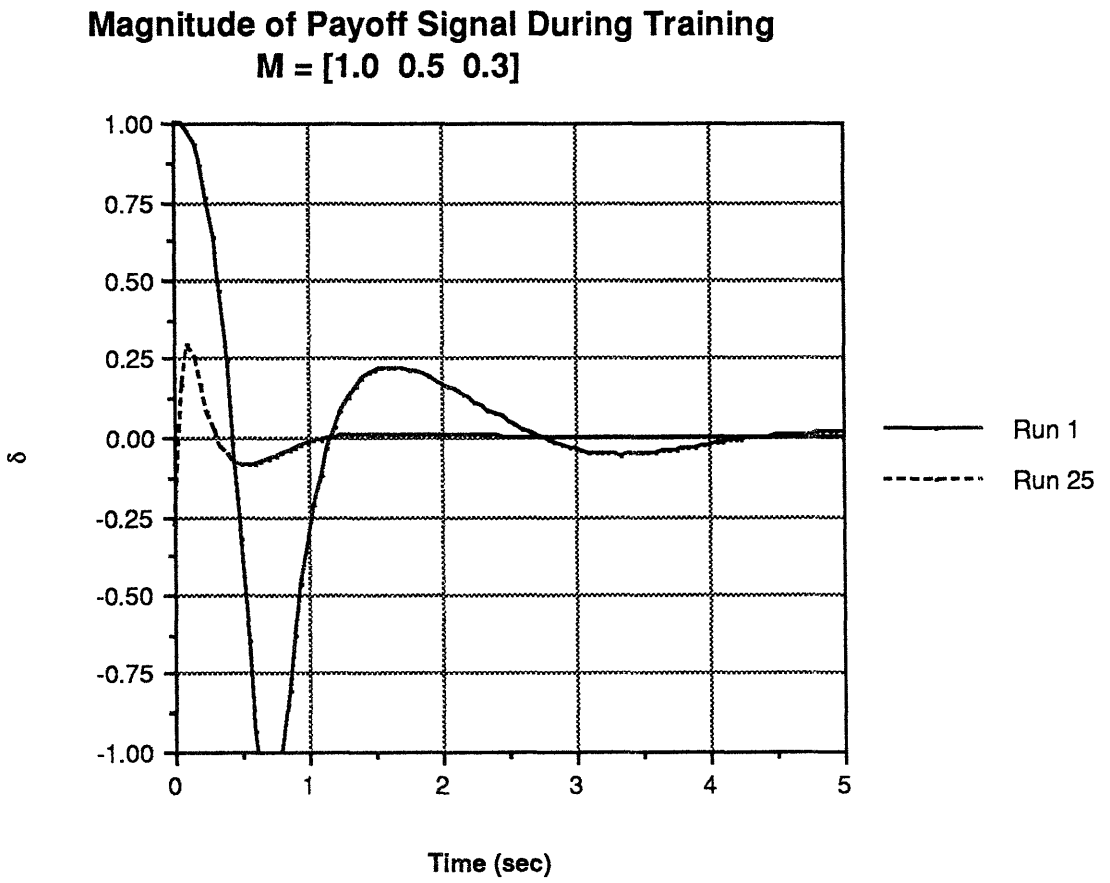

Figure 3.3.16: Magnitude of payoff signal during training phases: $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$

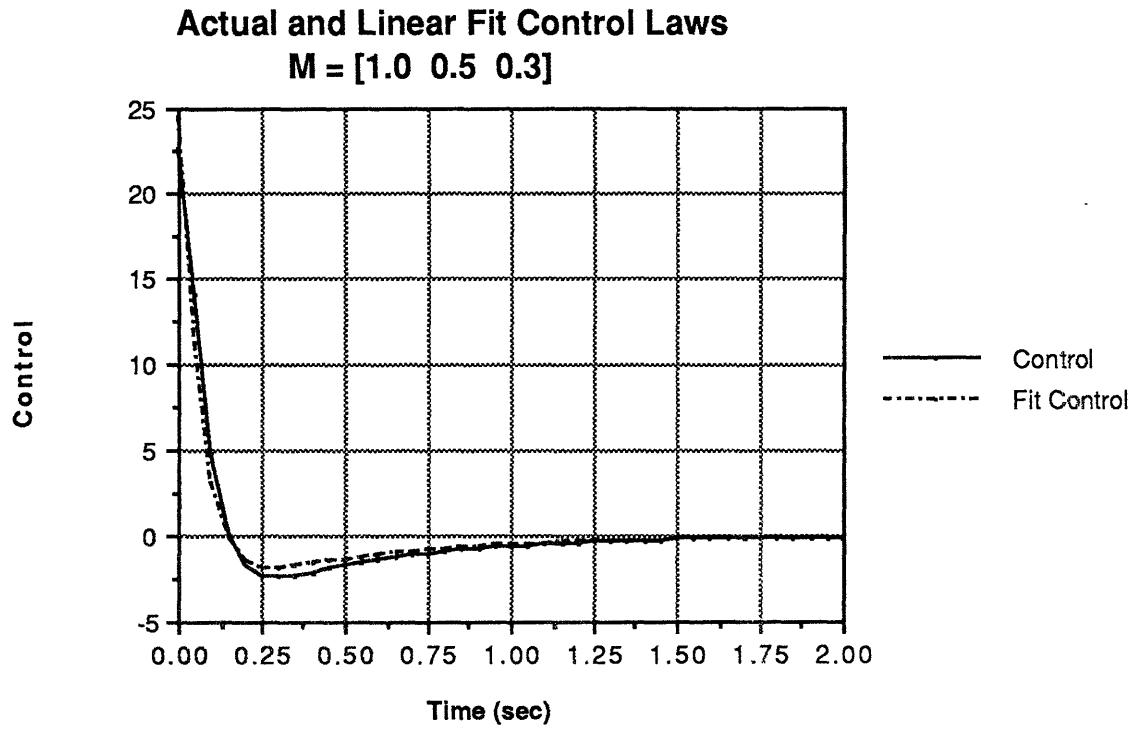

Figure 3.3.17: Actual and linear fit control laws: $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$

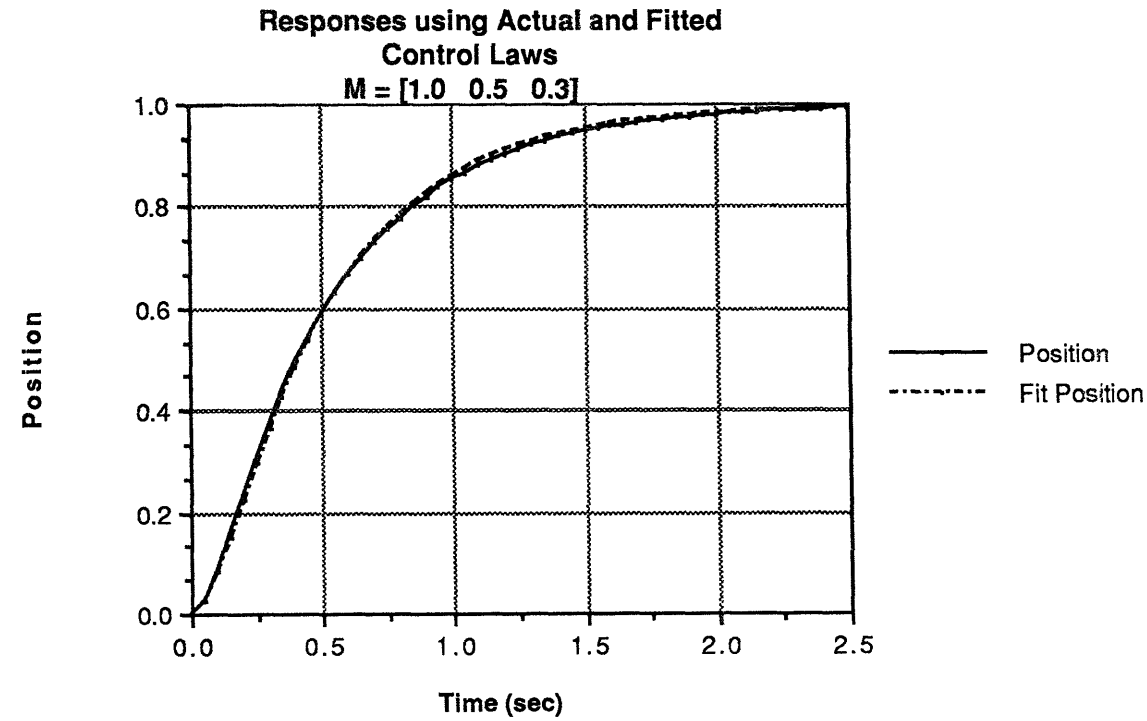

Figure 3.3.18: Plant responses using actual and linear fit controllers

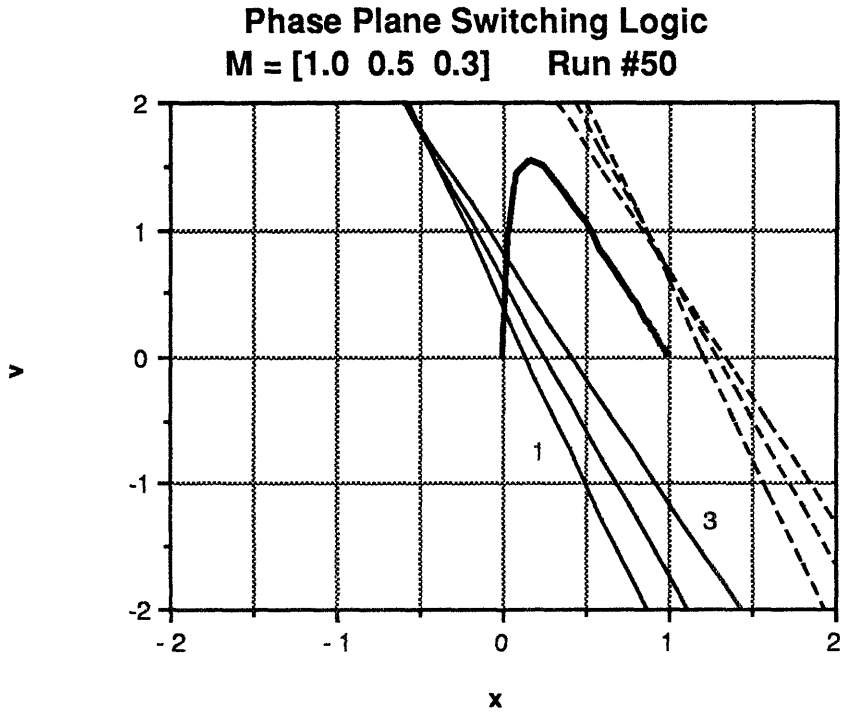

Figure 3.3.19: Phase space switching lines: $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$

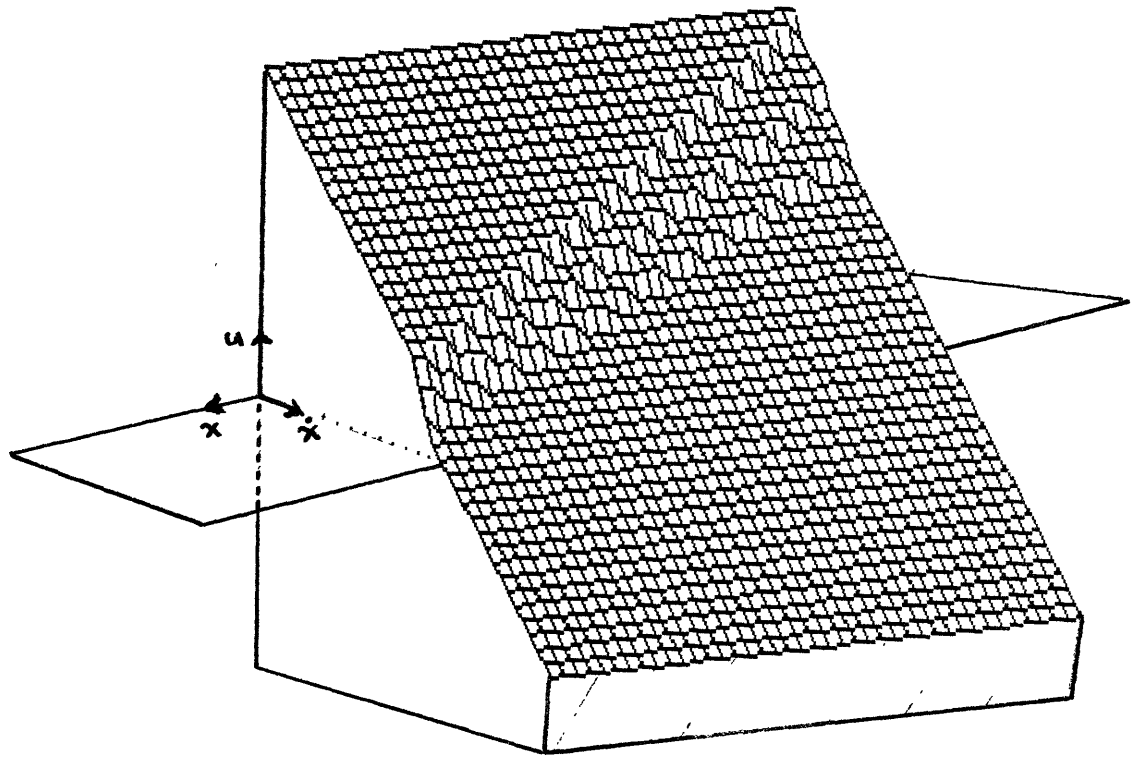

Figure 3.3.20: Control "surface" resulting from network control law: $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$

These observation have two implications. First, the response of the system is linear, but with a much higher effective bandwidth than would be expected from just the linear term of the controller. Second, control authority is distributed throughout the network: all of the neurons contribute to the control action during the simulation, and, from the steady state control analysis above, the active involvement of all neurons is required to maintain the plant at its equilibrium position. This distributed approach to the solution of the problem is characteristic of of neural network designs and contributes to the robustness and "graceful degradation" properties discussed earlier. One can see how these properties might manifest in this network. Destroying any of the hidden neurons or their biases would result in 1.) a controller with a slightly lower bandwidth, and 2.) a steady state equilibrium point slightly offset from the desired point (since now $0 \neq f_L(\mathbf{x}_d) + f_N(\mathbf{x}_d)$ because of the change which would result in $f_N(\cdot)$). The controller would not fail catastrophically when damaged, but would instead become gradually impaired. This will be demonstrated in greater detail in Section 3.5 below.

3.3.5 Increased Position to Velocity Weighting

In the previous examples the control laws developed by the NMC algorithm were almost completely linear over the range of plant states experienced during the simulation. Clearly, this will not always be the case and in this section results are presented which not only demonstrate instances of nonlinear controllers, but show that these represent sensible uses of the degrees of freedom in the network. Similar examples can also be found in Section 3.5.

Increasing the ratio of M_x to $M_{\dot{x}}$ produces the anticipated results in the solo responses seen after fifty runs. For higher ratios, the response becomes underdamped and faster, and for lower ratios the response is more heavily damped and slower, as Figure 3.3.21 demonstrates. This is not surprising, however the way in which these responses are accomplished is quite interesting. Figure 3.3.22 shows the control used to generate each of the responses shown in the previous figure. The $M_x = 1.0$ case is the canonical run analyzed in the previous section. Notice that for higher values of position weighting, the first derivative of the control becomes increasingly discontinuous at the cusp of the curve. Given that the $M_x = 1.0$ run can be fit almost exactly by a linear control law, one expects that, for example, the $M_x = 3.0$ run would not be well fit by a linear approximation.

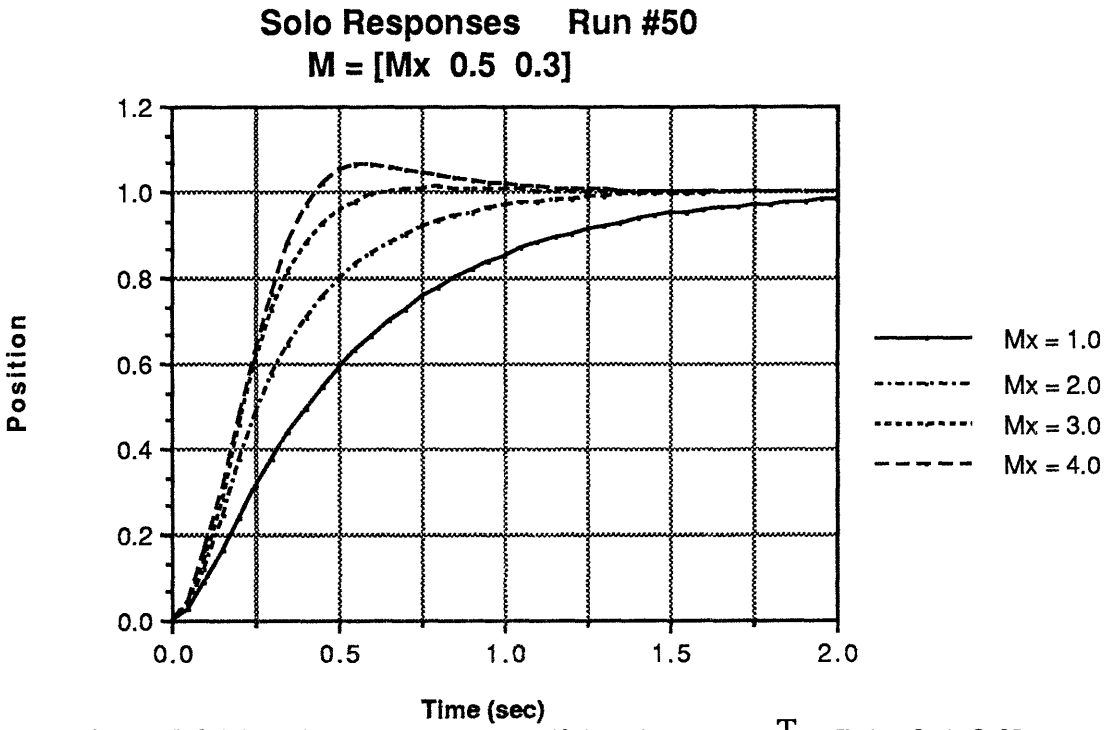

Figure 3.3.21: Plant responses on 50th solo run: $\mathbf{m}^T = [M_x \;\; 0.5 \;\; 0.3]$

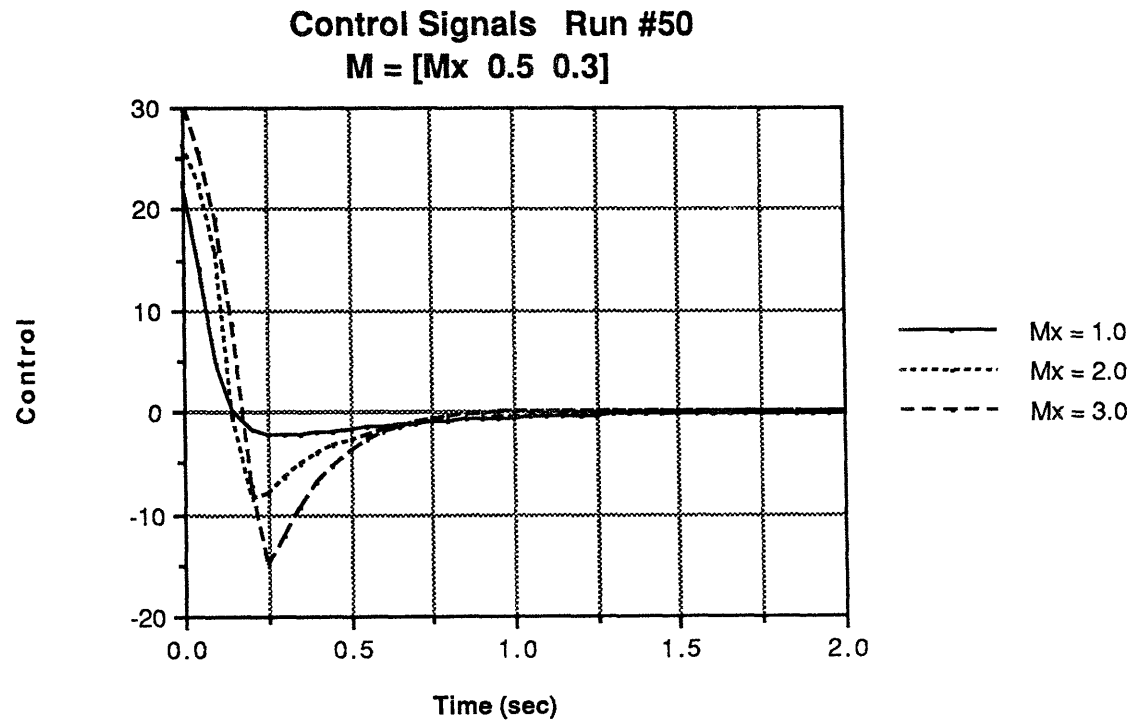

Figure 3.3.22: Control signals during solo runs: $\mathbf{m}^T = [M_x \;\; 0.5 \;\; 0.3]$

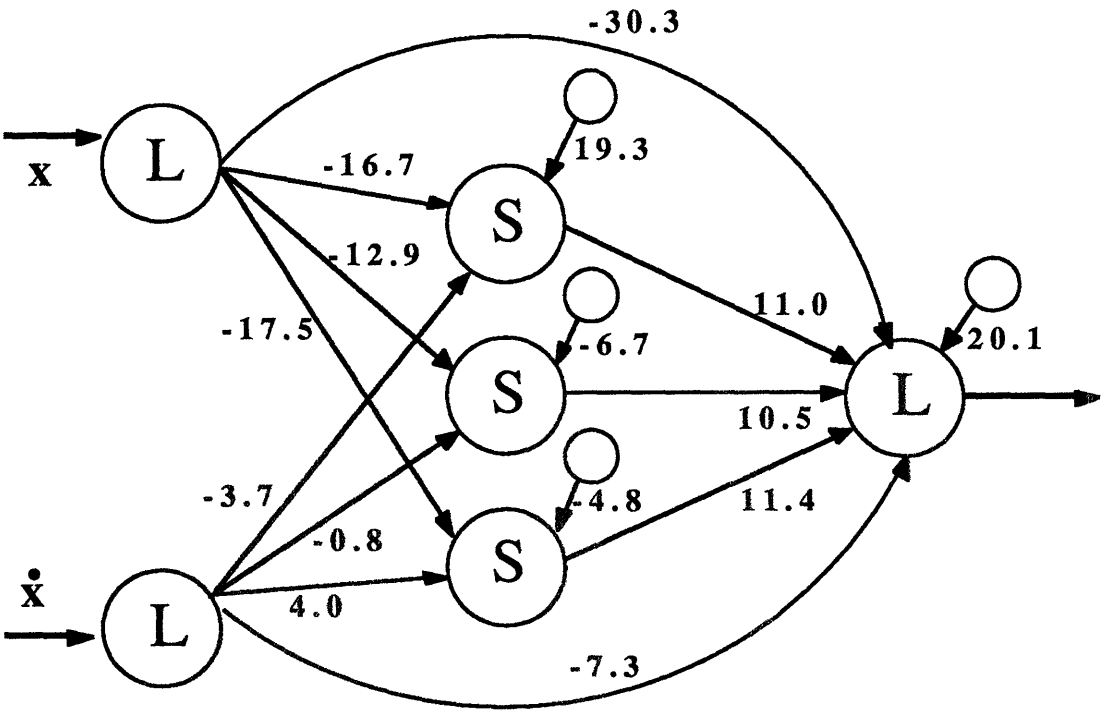

Figure 3.3.23: Network configuration after 50 iterations: $\mathbf{m}^T = [3.0\ \ 0.5\ \ 0.3]$

Figure 3.3.22 shows the network which results after the fiftieth iteration with $\mathbf{m}^T = [3.0\ \ 0.5\ \ 0.3]$. From this diagram, the control law can be written as

$$u = 20.1 - 30.3x - 7.3\dot{x} + 11.0\sigma_2 + 10.5\sigma_3 + 11.4\sigma_4 \qquad (3.14)$$

where,

$$\begin{aligned} \sigma_2 &= \text{sig}(19.3 - 16.7x - 3.7\dot{x}) \\ \sigma_3 &= \text{sig}(-6.7 - 12.9x - 0.8\dot{x}) \\ \sigma_4 &= \text{sig}(-4.8 - 17.5x + 4.0\dot{x}) \end{aligned} \qquad (3.15)$$

Notice that the all the weights associated with the input neuron which encodes position are substantially increased from the canonical weights, equation (3.10), while those of the input neuron which encodes velocity are hardly affected.

Figure 3.3.23 shows the results of trying to fit a linear control law to the experimental data. The best approximation gives:

$$u = 24.6(1.0 - x) - 13.4\dot{x} \qquad (3.16)$$

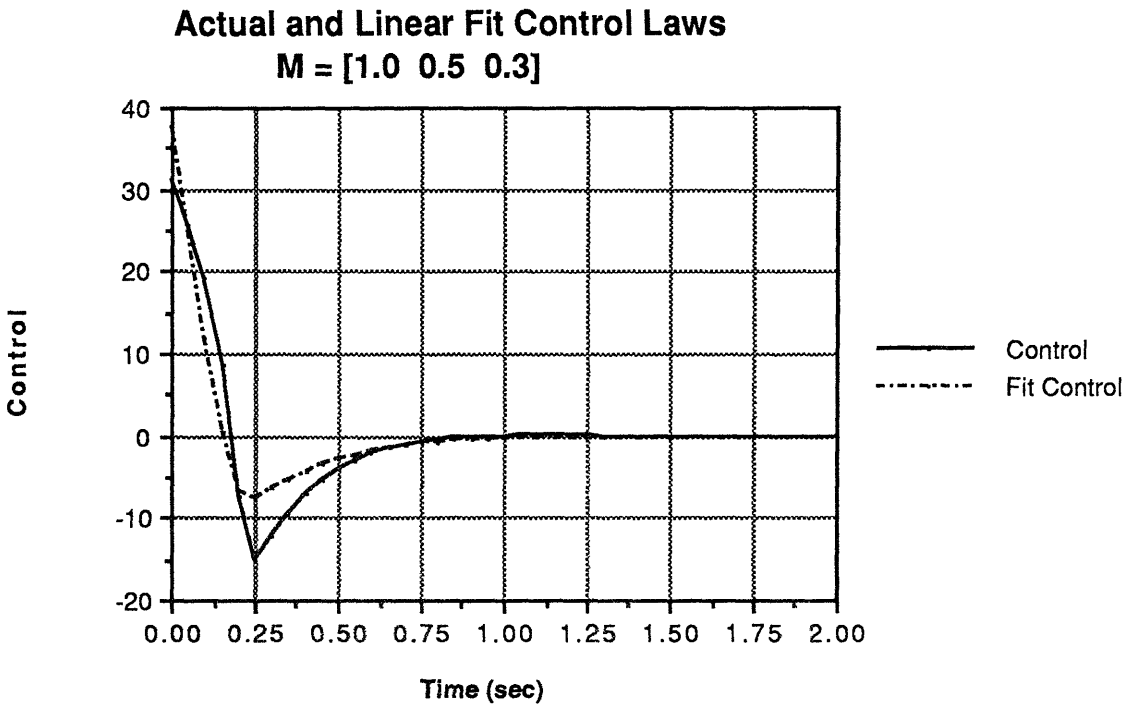

Figure 3.3.24: Best fit linear control law for $\mathbf{m}^T = [3.0 \;\; 0.5 \;\; 0.3]$

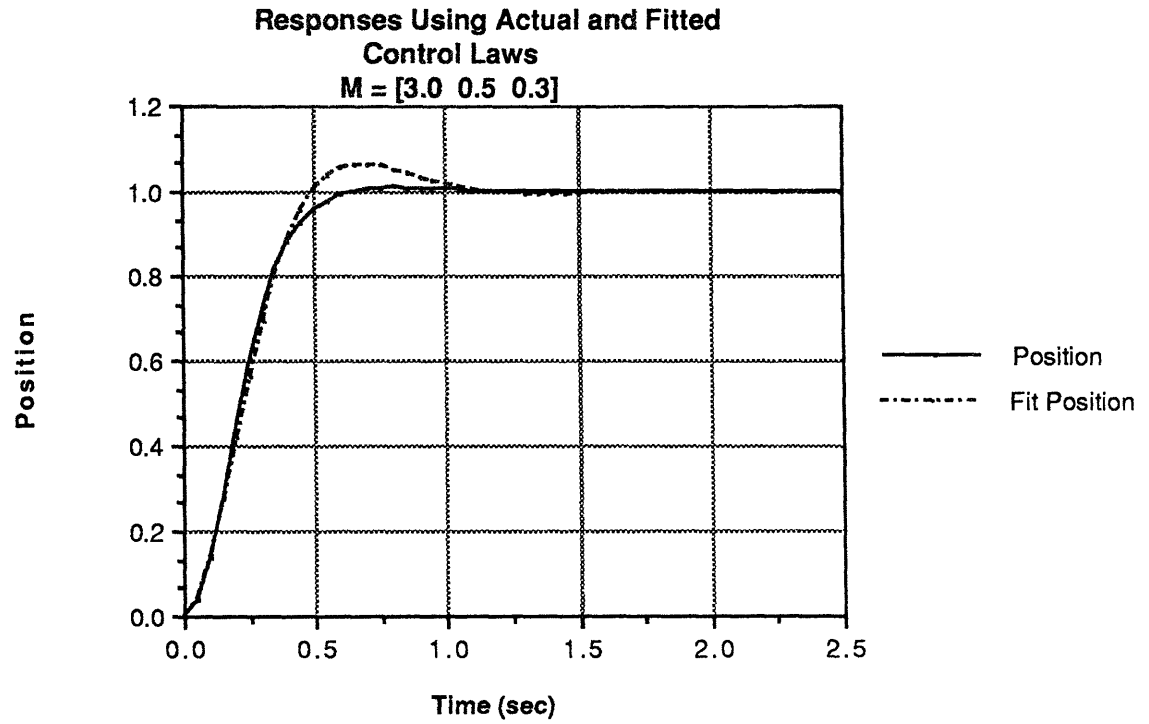

Figure 3.3.25: Comparison of linear fit and actual responses; $\mathbf{m}^T = [3.0 \;\; 0.5 \;\; 0.3]$

which is clearly inadequate as can be seen from the figure. The agreement is very good at the beginning and ending of the control profile. The major discrepancy seems to be the large spike of negative control which occurs at t = 0.25 seconds; this occurs slightly later and with much larger magnitude than would be predicted by the linear law. Plotting the responses generated using the fitted and actual control laws (Figure 3.3.25) reveals that the effect of this negative spike is to damp the overshoot of the response, from almost 10% for the linear model, to under 2% for the actual control law.

The network thus uses some of its degrees of freedom to damp the overshoot of the closed loop response in a nonlinear fashion. Exactly how this is done will be examined shortly, first it is necessary to understand *why* the network has implemented a controller of this form, especially given that a heavily damped response can be generated easily using just linear feedback. From Figure 3.3.21, the settling time of the closed loop response using the actual control law is about 0.5 seconds with an overshoot of about 2%; this is characteristic of a set of closed loop poles at $s_{1,2} = -5 \pm 4j$. A linear controller which created a closed loop system with these poles from the double integrator open loop plant would require a maximum of +41 units of control (at t = 0) to take the system from rest to x_d; the NMC network accomplishes the same response with a maximum of +31 units of control, also at t = 0 seconds. It would thus appear the the observed control law has arisen as a result of the tradeoff performed by the algorithm between maximum control authority and speed of the closed loop response. In a sense, the additional position weighting has told the NMC to speed up the response, faster than the canonical example; since this could not be accomplished using the linear control scheme developed in the canonical experiment because it would require too much control, the algorithm has begun to make use of its other degrees of freedom. How much control is "too much" will be quantified below.

Recall that the rise time for a second order system is, roughly, inversely proportional to the damping ratio and directly proportional to the natural frequency. For values of damping greater than about $\zeta = 0.707$, the (5%) settling time is not much longer than the rise time; for damping ratios less than this the settling time becomes appreciably longer. Within certain bounds, it is possible to keep the rise time of the system the same with a lower natural frequency if one also decreases the damping ratio. For a linear feedback control scheme, this would reduce the amount of maximum control required, but it would also increase the overshoot of the closed loop response, and probably lengthen the settling time. However, the NMC is not linear! Thus, it can deliberately "target" a closed loop system with approximately the same rise time, but with a lower damping ratio and natural frequency than the response it emulates; it is this target trajectory which is shown in

Figure 3.3.25. This allows it to use less control when the system starts from rest, and then strategically introduce extra negative control in such a way as to "jam on the brakes" and stop the overshoot which might otherwise develop, thus also reducing the settling time. This is a very reasonable way to use the degrees of freedom in the controller.

Figure 3.3.26 shows the switching lines which implement the controller. Notice how these differ from the canonical run, Figure 3.3.19. In particular, the on/off lines of different neurons now have different slopes, and the linear regions are of different widths. Neuron number four is clearly off for all values of the state encountered in this simulation. A critical point seems to exist where the switching lines of neurons number three and five intersect; indeed the fact that the trajectory of the system from rest to equilibrium passes through this point also suggests its importance.

This figure, together with a plot of neural activity versus time, Figure 3.3.27, provides a complete picture of what occurs inside the network as the plant state evolves toward equilibrium. As the trajectory approaches the critical point, $\mathbf{x}_c^T = [0.25\ 3.25]$, neuron number five turns on, briefly adding +11.4 (see Figure 3.3.23) units of control to the plant. When the critical point is reached, neurons three and five both begin to rapidly shut off, forcing the control more negative by approximately 20 units (neuron five turns off removing its 11.4 units, and neuron three reduces to one quarter output, removing a further 8.6 units); this, combined with the linear contribution of the controller, accounts for the large negative spike seen in the control. After the critical point, neurons three and four remain off, and neuron five continues to function now in its linear region, coming almost completely on by the time the plant has settled to equilibrium. The switching logic implemented by the network has thus allowed it to design a nonlinear controller which produces a rapidly responding, well damped, closed loop response with a limited amount of maximum control authority.

Despite the fact that the NMC has been somewhat "clever" in designing these switching lines to obtain a fast, heavily damped response which does not require much control, this "ideal" solution will only be seen when the plant starts from rest. The critical point seen in the switching line configuration would not, in general, be encountered by the state trajectory if the plant were released from a nonzero initial condition. The response seen in this case would tend more to emulate the more lightly damped "target" trajectory as the plant settles. This certainly makes sense; the NMC is essentially learning by trial and error, it has no reason to design the switching lines for state configurations it has never encountered during its training phases. Again, this reveals one of the weaknesses in the way these experiments were conducted, and will be discussed further in Section 4.1.5.

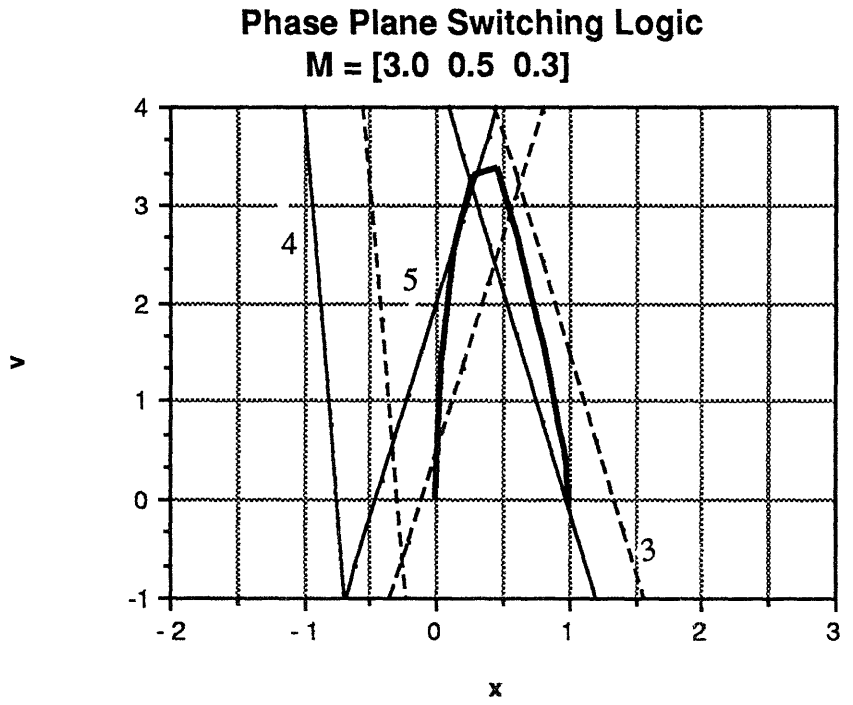

Figure 3.3.26: Neuron switching logic; $\mathbf{m}^T = [3.0 \;\; 0.5 \;\; 0.3]$

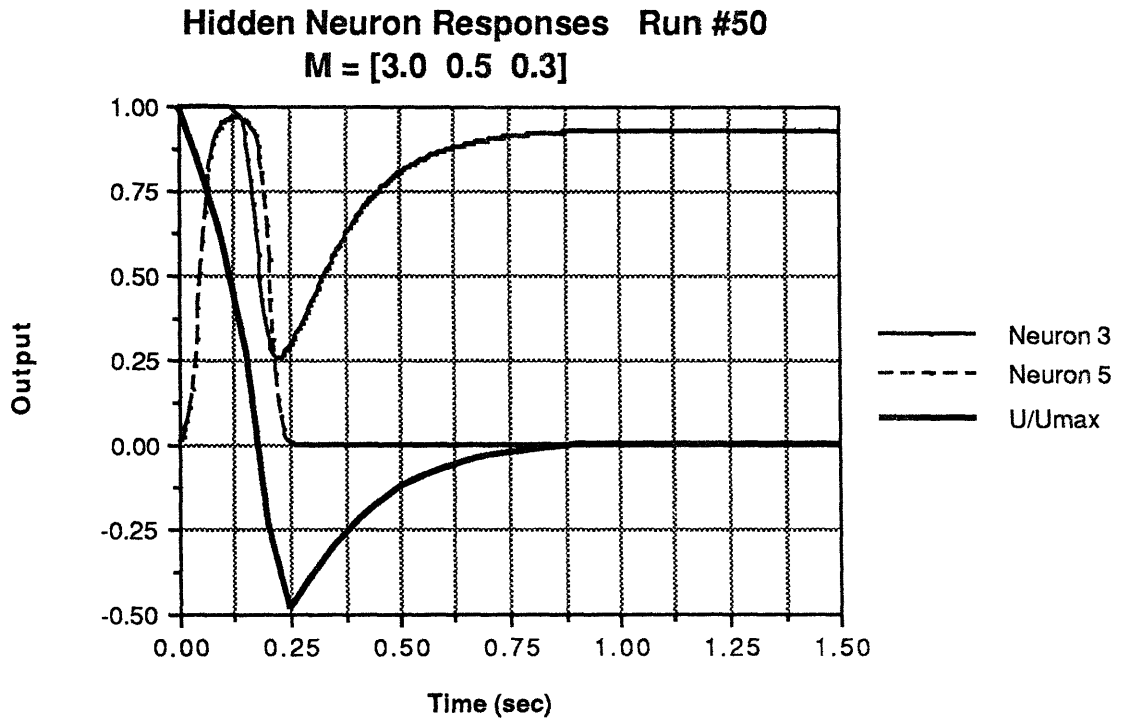

Figure 3.3.26: Hidden neural activity during simulation; $\mathbf{m}^T = [3.0 \;\; 0.5 \;\; 0.3]$

3.3.6 Effect of the control weighting exponent

The effect of variations of M_u in the control contribution to the payoff function has already been discussed in Section 3.3.3 above. In this section, the impact of varying the exponent in this term is evaluated. Figure 3.3.28 shows that the shape of the closed loop response generated on the 50th solo run is virtually identical for several different values of n. However, as Figure 3.3.29 shows, the maximum control used after the learning has stabilized is progressively lower with increasing n.

It has already been noted above that the ideal solution for the network is to devise a control signal which makes $\delta(t) \equiv 0.0 \; \forall t$; i.e. which makes the optimal tradeoff between position and velocity deviations, and control usage. Judging from the preliminary results analyzed for the canonical run (Figure 3.3.16), there is reason to believe that this is just what the algorithm is doing. Accepting, for the moment, that this tradeoff can be make *exactly* for the instant the plant starts from rest (before the dynamics of the plant begin to assert themselves), and noting that the form of the controllers devised by the network so far apply the maximum control at precisely this instant, $t = 0.0$, a good approximation can be developed to predict the maximum control which the network will use after the training has stabilized, for the second order plants under consideration:

$$u_{max} \cong u_{ult}\left(\frac{M_x}{M_u}(x_d - x(0))\right)^{1/n} \tag{3.17}$$

For the canonical run of Section 3.3.3, this equation would yield $u_{max} \cong 21.6$, which compares favorably with the values observed after fifty solo runs of $u_{max} = 22.7$. Figure 3.3.30 summarizes the predicted versus the observed u_{max} for a variety of M_x, M_u, and n. Note that the estimates tend to be low by from 2-20%. However, recall from Figure 3.3.13 that fifty runs was not quite enough for the control law to converge to its final value, in fact the maximum control used in that case fell by 8% from the fiftieth to the 250th run. When the predicted value for the runs plotted in Figure 3.3.13 (for which $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.7]$), is instead compared with the value seen on the 250th run, the error is less than 0.2% (17.5 vs 17.3). The errors in the other estimates would similarly be expected to improve as a function of the number of training sequences the network has undergone.

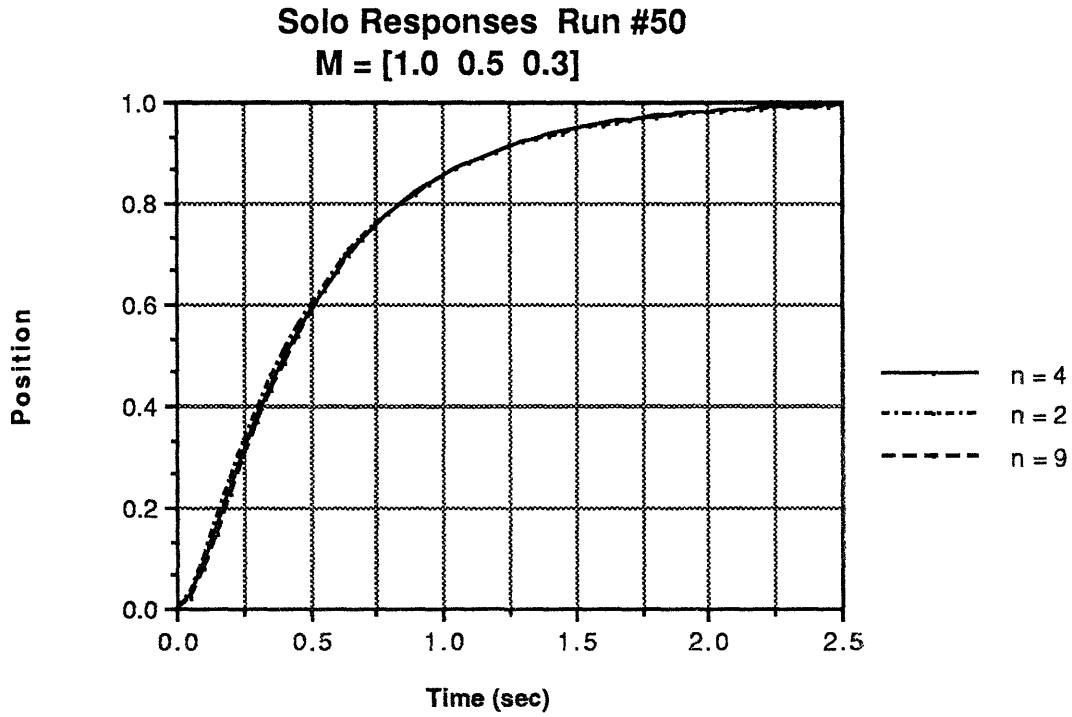

Figure 3.3.28: Variation of responses with increasing n; $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$

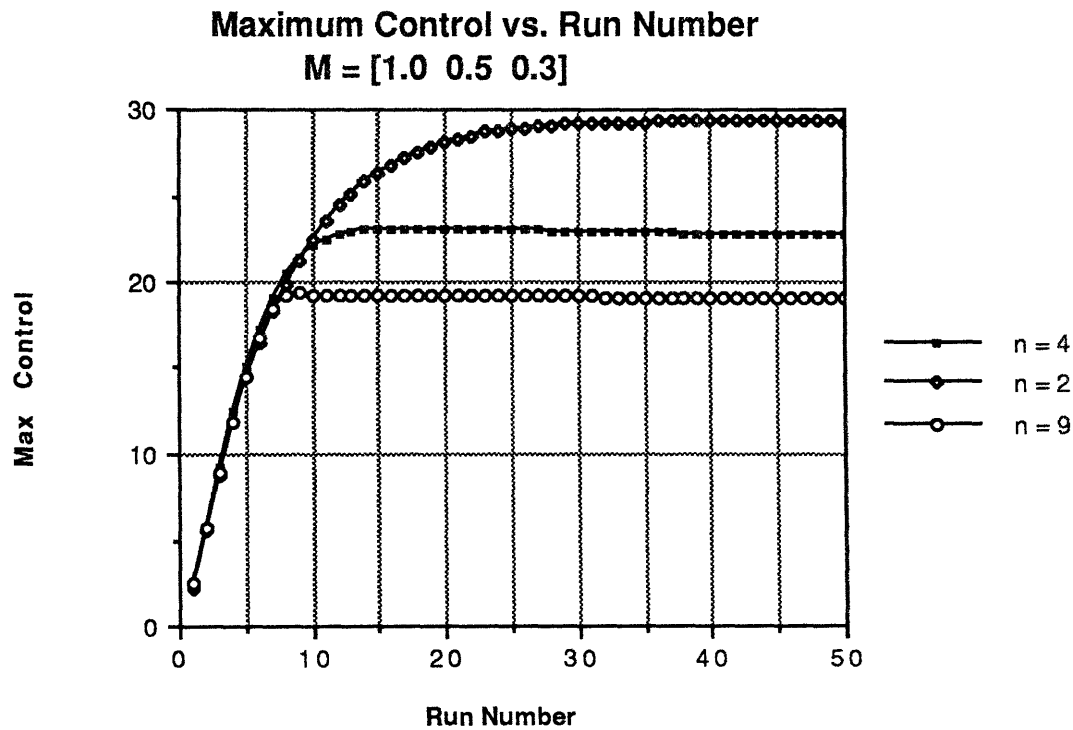

Figure 3.3.29: Variation of maximum control with increasing n; $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$

M_x	M_u	n	Predicted u_{max}	Actual u_{max}
1.0	0.3	2	29.2	29.4
1.0	0.3	4	21.6	22.7
1.0	0.3	7	19.0	19.9
1.0	0.3	9	18.3	19.0
1.0	0.7	4	17.5	18.7
2.0	0.3	4	25.7	26.4
3.0	0.3	4	28.5	31.1
4.0	0.3	4	30.5	36.2

Figure 3.3.30: Predicted (equation 3.17) vs. observed u_{max}

3.3.7 Variation of Network Parameters

The parameters α, η, and f in the NMC algorithm (Figure 2.6.4) represent properties of the network itself, as opposed to those of the payoff function examined above. This section analyzes the effect of changes in these network parameters on the ability of the NMC to converge to a stabilizing control law.

The algorithm is actually surprisingly insensitive to changes in most of these parameters. From Figure 3.3.31 it is evident that changing the learning rate, η, does not significantly change the responses obtained by the network after fifty runs. Interestingly, however, while $\eta = 0.1$ results in the most severe transients during the initial learning phase, as indicated in Figure 3.3.32(a), it also results in the most rapid convergence to the final form of the closed loop response, as Figure 3.3.32(b) and (c) show, although the responses for the different η are indistinguishable by the fifth solo run.

Changing the decay rate, α, similarly does not alter the form of the solution devised by the network, as shown in Figure 3.3.33. However, higher values of α cause the training phase to become progressively more unstable, as shown in Figure 3.3.34. In fact, for values of α higher than about 0.875, the entire training sequence becomes unstable and the network can never develop a stabilizing controller. This is not surprising in light of equation (2.9); the weight update equation is essentially a first order finite difference equation with a "time constant" of α. While this parameter has an important damping effect in the search of the weight space, for values of α close to 1.0 one would expect to see the

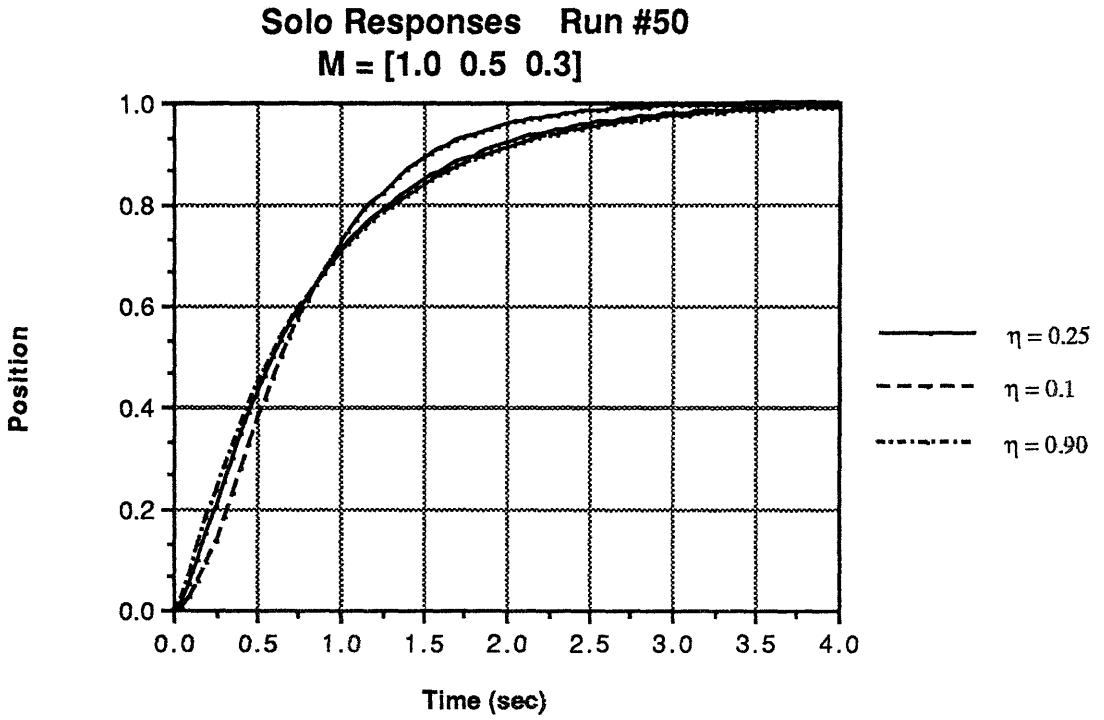

Figure 3.3.31: Variation in responses with increasing η: $\mathbf{m}^T = [1.0\ 0.5\ 0.3]$

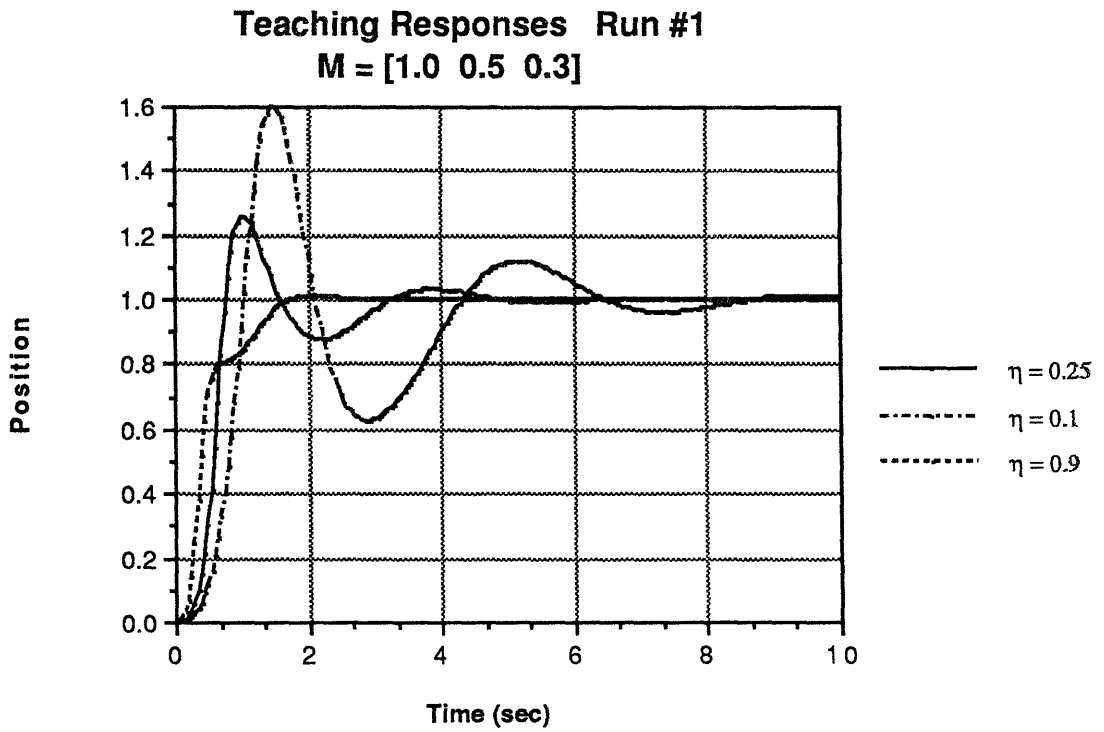

Figure 3.3.32(a): First training responses with increasing η: $\mathbf{m}^T = [1.0\ 0.5\ 0.3]$

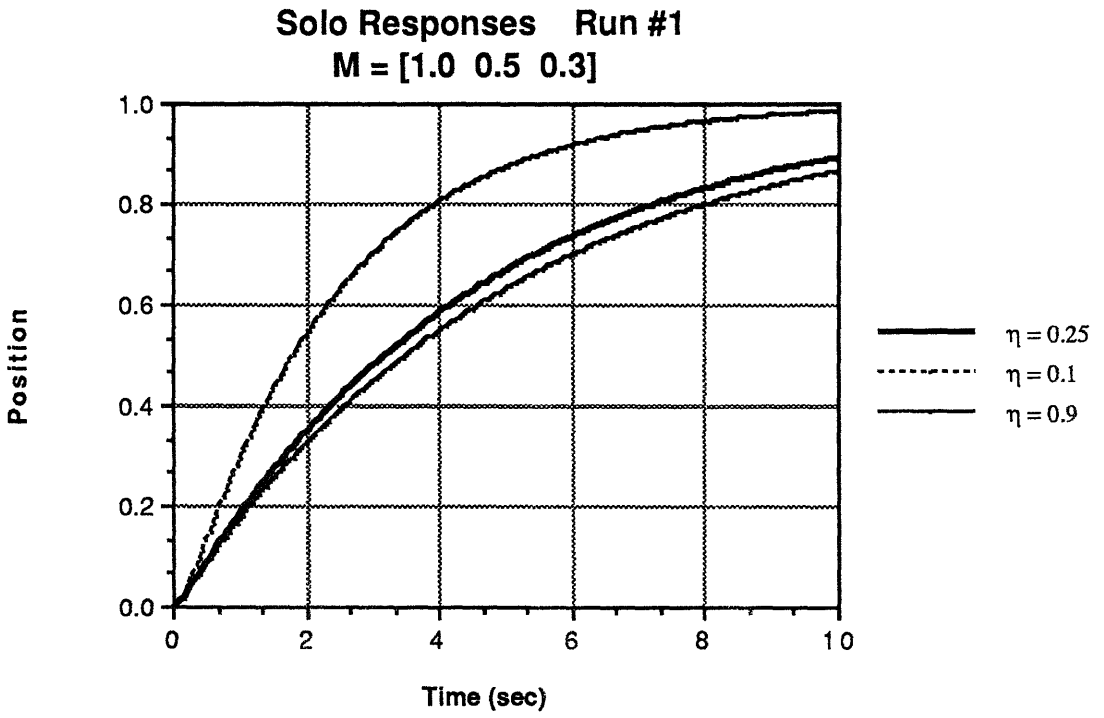

Figure 3.3.32(b): First solo responses with increasing η: $\mathbf{m}^T = [1.0 \ 0.5 \ 0.3]$

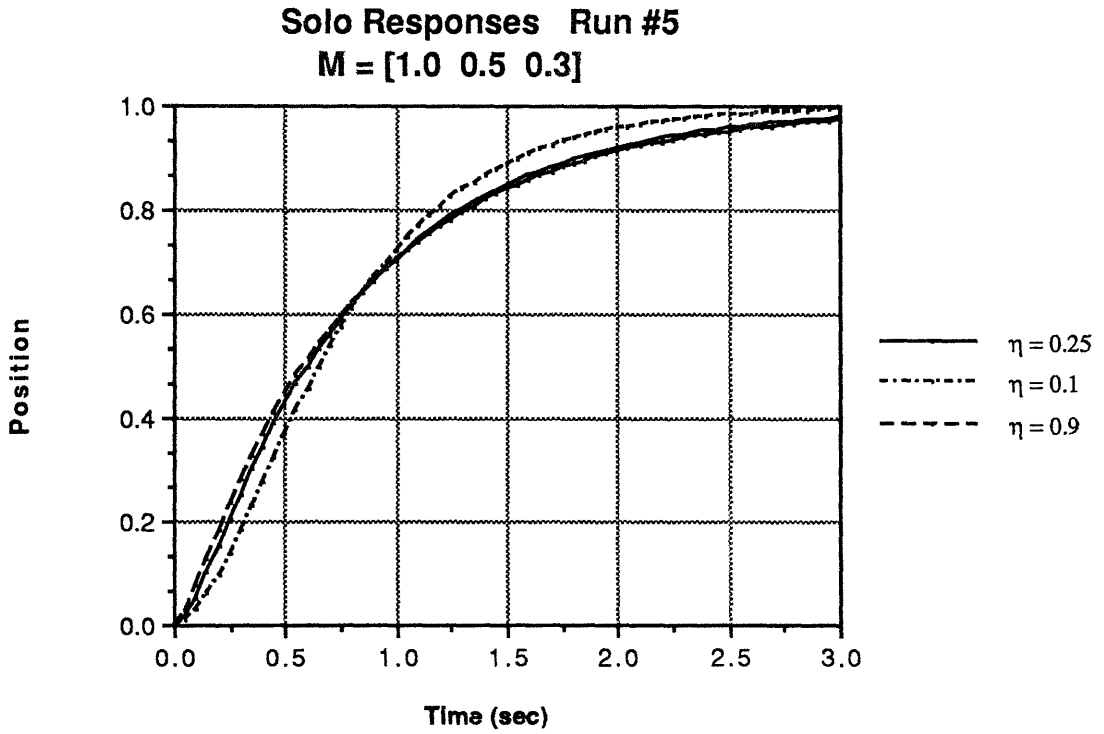

Figure 3.3.32(c): Fifth solo responses with increasing η: $\mathbf{m}^T = [1.0 \ 0.5 \ 0.3]$

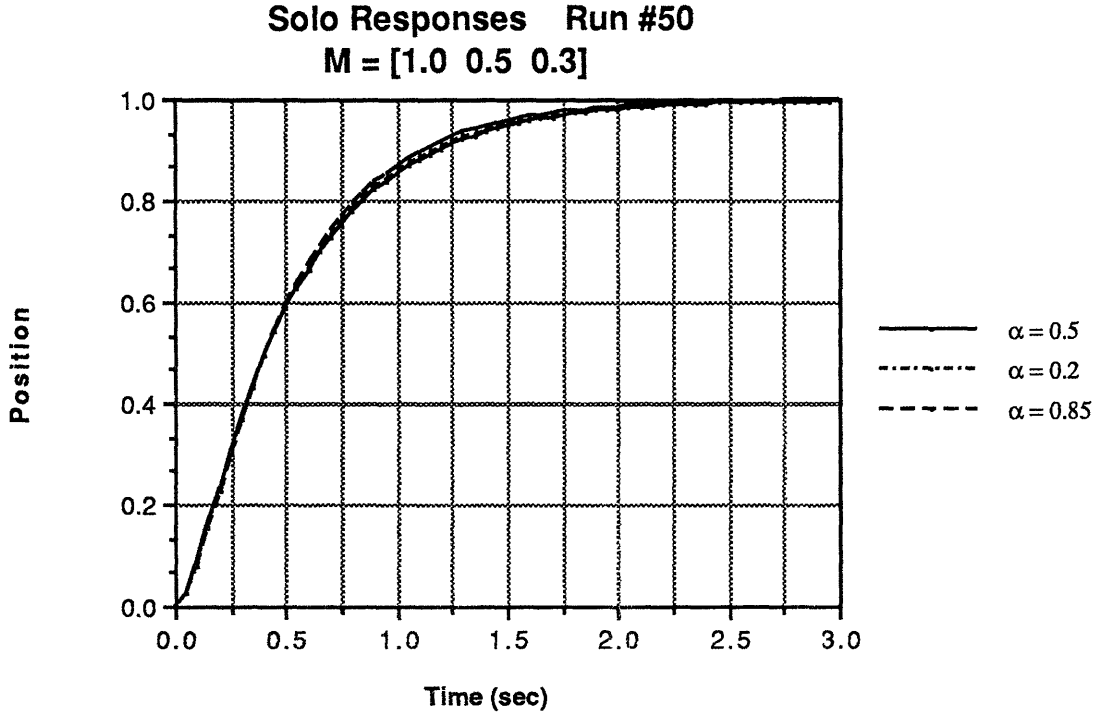

Figure 3.3.33: Variation in responses with increasing α: $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$

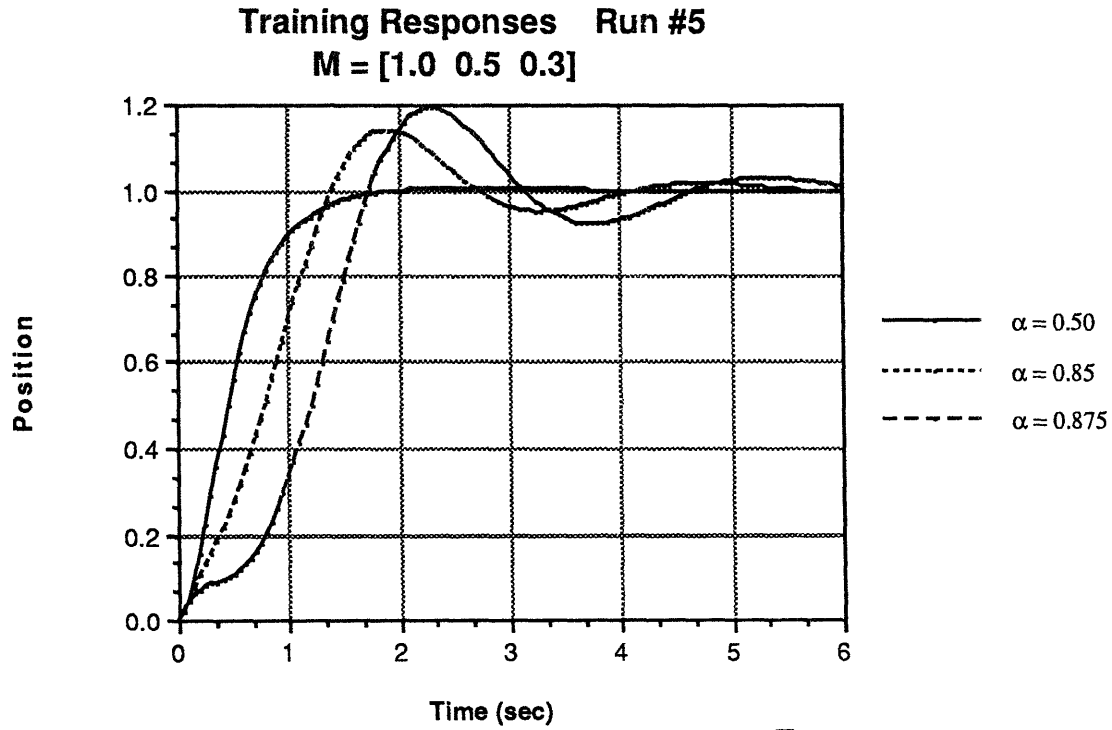

Figure 3.3.34: Training responses for increasing α: $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$

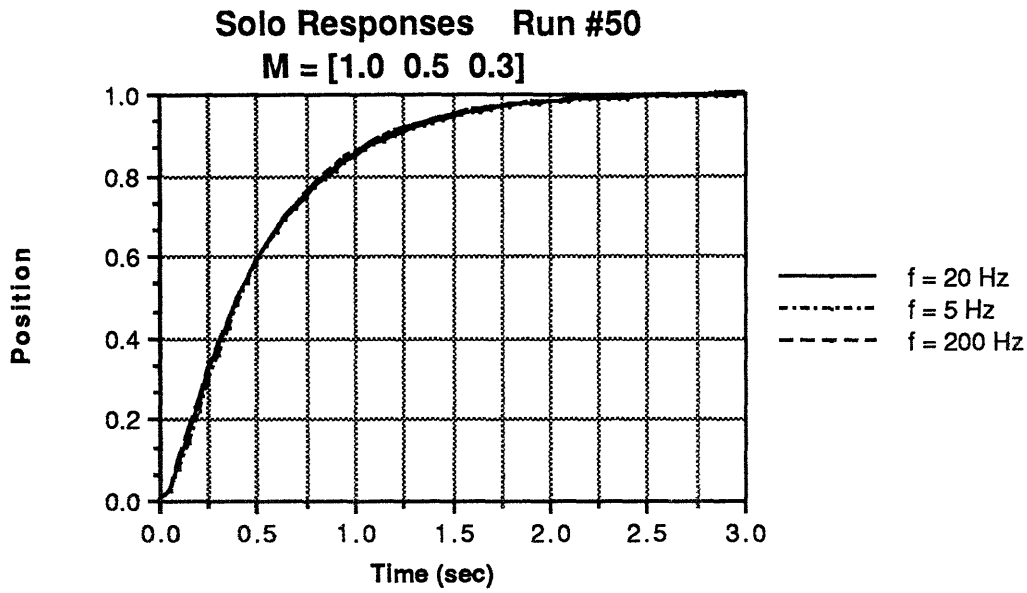

Figure 3.3.35: Variation of responses with increasing f: $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$

network become increasingly "sluggish" and unable to respond quickly enough to changes in the teaching stimuli, with the result that the plant would go unstable. Exactly why the algorithm becomes unstable at this particular value of α is not known, but it is clear that moderate values of α $(0.1 \leq \alpha \leq 0.6)$ will yield the best results.

Changing the frequency of the teaching iterations also has minimal effect on the solutions devised by the network on the 50th run, as Figure 3.3.35 demonstrates. This is very encouraging from the standpoint of real time applications of the NMC algorithm where teaching iterations, because they are so computationally intensive, may have to be limited.

The rate at which the network is taught does, however, have an effect on how quickly the network arrives at its steady state control law, as shown in Figures 3.3.36(a) and (b). Again, it appears that it is the *slower* teacher which converges most rapidly to the final control law. This makes some intuitive sense: the slower the learning, the larger the excursions from the desired equilibria which will occur during the training phase, as demonstrated in Figure 3.3.36(c). This is the same phenomenon which was observed in the experiments in which η was lowest. It would appear that, since slower learning networks experience more of the total state space and encounter payoff signals of higher magnitude while they train, they converge in fewer iterations

On the basis of the above analysis of the parameter space, nominal values of $\alpha = 0.5$, $\eta = 0.25$, and $f = 20$ Hz were selected for use in all of the following experiments.

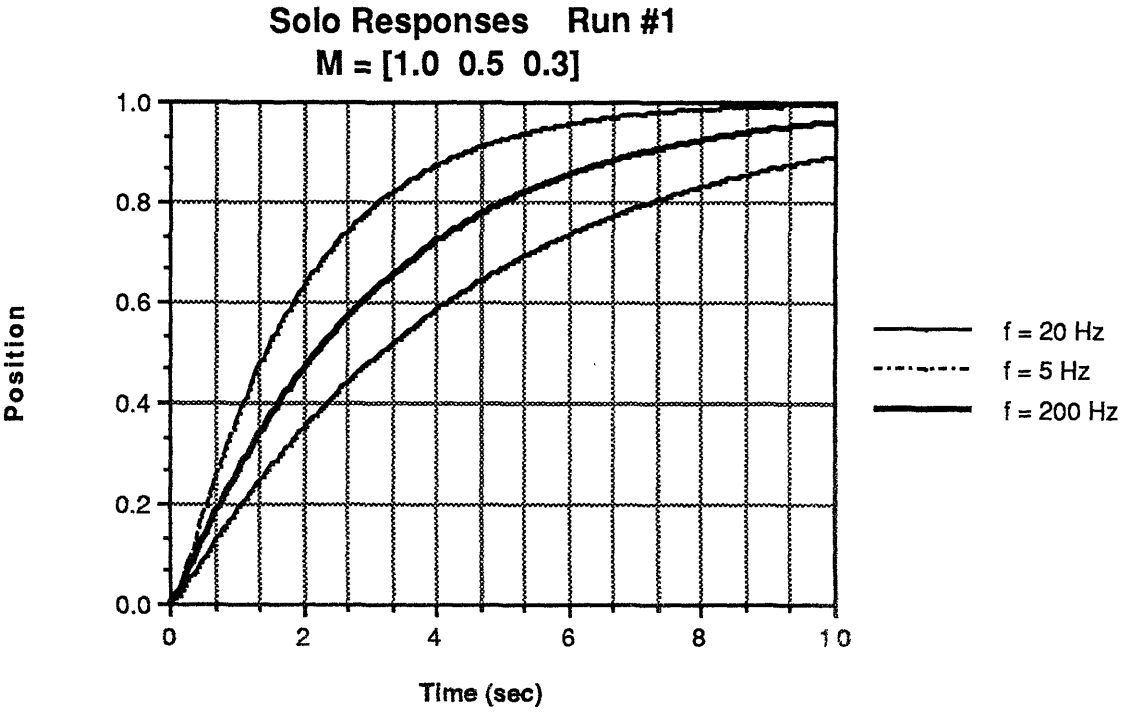

Figure 3.3.36(a): Responses after one training phase, increasing f:
$\mathbf{m}^T = [1.0 \quad 0.5 \quad 0.3]$

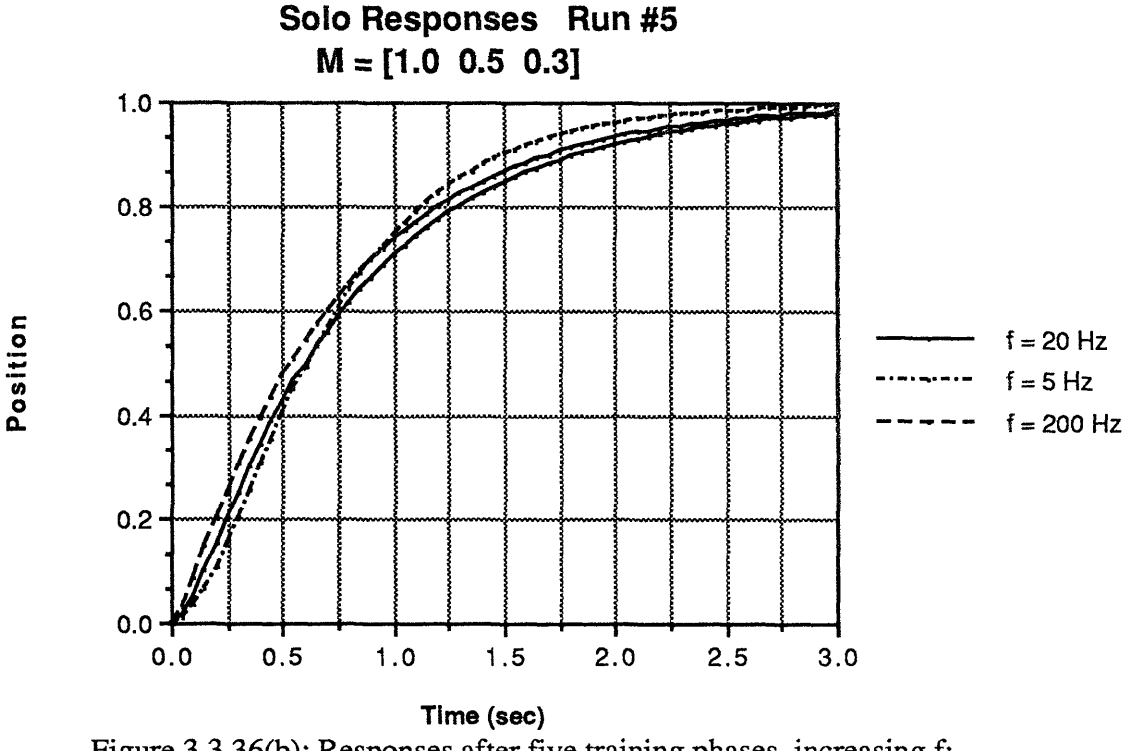

Figure 3.3.36(b): Responses after five training phases, increasing f:
$\mathbf{m}^T = [1.0 \quad 0.5 \quad 0.3]$

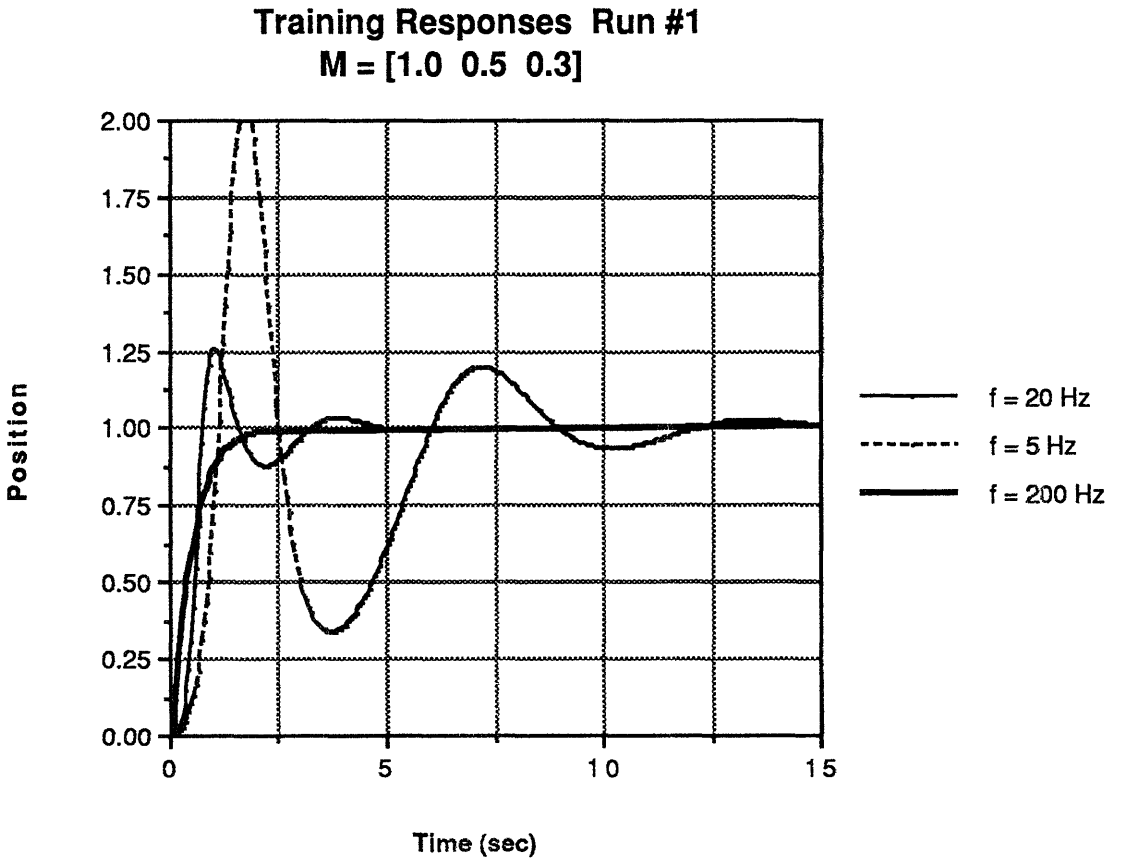

Figure 3.3.36(c): First training responses, increasing f : $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$

3.4 The Model Reference Payoff Function

The next experiments sought to evaluate the effectiveness of the model reference form of the payoff function. For a second order system, equation (2.X) becomes:

$$\delta(t) = M_x(x_d(t) - x(t)) + M_{\dot{x}}(\dot{x}_d(t) - \dot{x}(t)) + M_u\left|\frac{u}{u_{ult}}\right|^n \qquad (3.18)$$

where here $x_d(t)$ and $\dot{x}_d(t)$ are the time varying model trajectories for the two states. Not that the only difference between this form of the payoff and that examined above is the time varying nature of the desired states. One of the side effects of this change is that the magnitude of $\delta(t)$ is generally somewhat lower during the training runs. For this reason, the weights chosen in equation (3.18) will be somewhat higher than those used for the canonical run of Section 3.3.3. The actual values of gains used will again be identified by $\mathbf{m}^T = [M_x \quad M_{\dot{x}} \quad M_u]$; values of $n = 4$, $\alpha = 0.5$, $\eta = 0.25$, and $f = 20$ Hz were used for all simulations.

Three different model trajectories were specified. The first two were first order trajectories corresponding to a pole at $s_1 = -0.5$ and $s_1 = -1.0$ respectively. Since these are first order models, there is no model second state to specify; the velocity weight for these experiments was thus set to $M_{\dot{x}} = 0.0$. The third trajectory was that of a second order underdamped oscillator corresponding to poles at $s_{1,2} = -1 \pm 2j$. For this third experiment, both position and velocity model trajectories were specified.

The problem of following the model trajectories seemed to be much more difficult than responding to the state weighted payoff function. To minimize $\delta(t)$ here the network must find *exactly* that sequence of controls which produces the desired responses in the state variables. This difference in difficulty with the previous experiments manifested when each model reference experiment required exactly one "failure" signalled by the trainer before stabilizing. This failure always occurred early in the first training phase.

3.4.1 First Order Model Trajectories

The first experiment had a model position trajectory specified by:

$$x_d(t) = 1 - \exp(-t/2) \qquad (3.19)$$

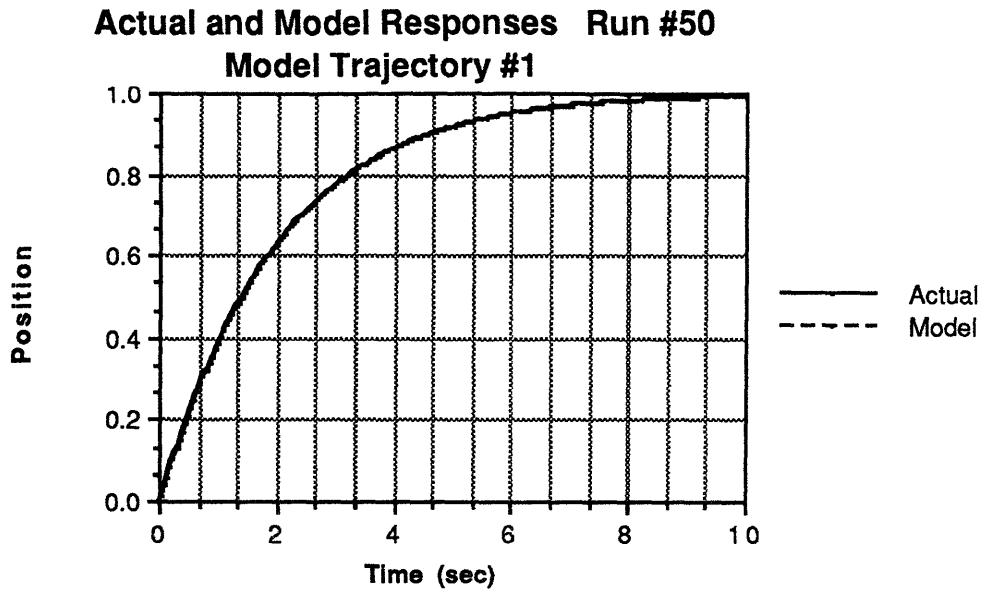

Figure 3.4.1: Comparison of actual and model trajectories for experiment #1

and no velocity reference. Accordingly, a weight vector of $\mathbf{m}^T = [2.0 \quad 0.0 \quad 0.3]$ was used. Figure 3.4.1 shows the model trajectory and the actual trajectory generated during the fiftieth solo run. Clearly the two responses are virtually identical.

Figure 3.4.2 shows the network which implements the control law and Figure 3.4.3 shows the switching lines which result. Notice immediately from this last figure that during the simulation all the hidden neurons are operating in saturation, meaning that the response of the plant will be governed by the linear terms of the controller. From inspection of Figure 3.4.2, this linear part is given by $u = -18.2x - 36.6\dot{x}$. Further, since neuron five operates in the on saturation region during the entire simulation while the other two neurons are always off, and since this neuron together with the control bias neuron contributes a total of +18.2 units of control, the complete control law, valid over the states visited during the simulation, is given by $u = 18.2(1 - x) - 36.6$. This is a perfect linear state feedback control law with its equilibrium at the desired final position for the plant! In fact, with this control law the closed loop poles lie at $s_1 = -0.51$ and $s_2 = -35.49$ which is exactly the response required (the pole corresponding to the evolution of the extra state has been "pushed" far enough into the left hand plane that its effects are negligible over the time scale of interest).

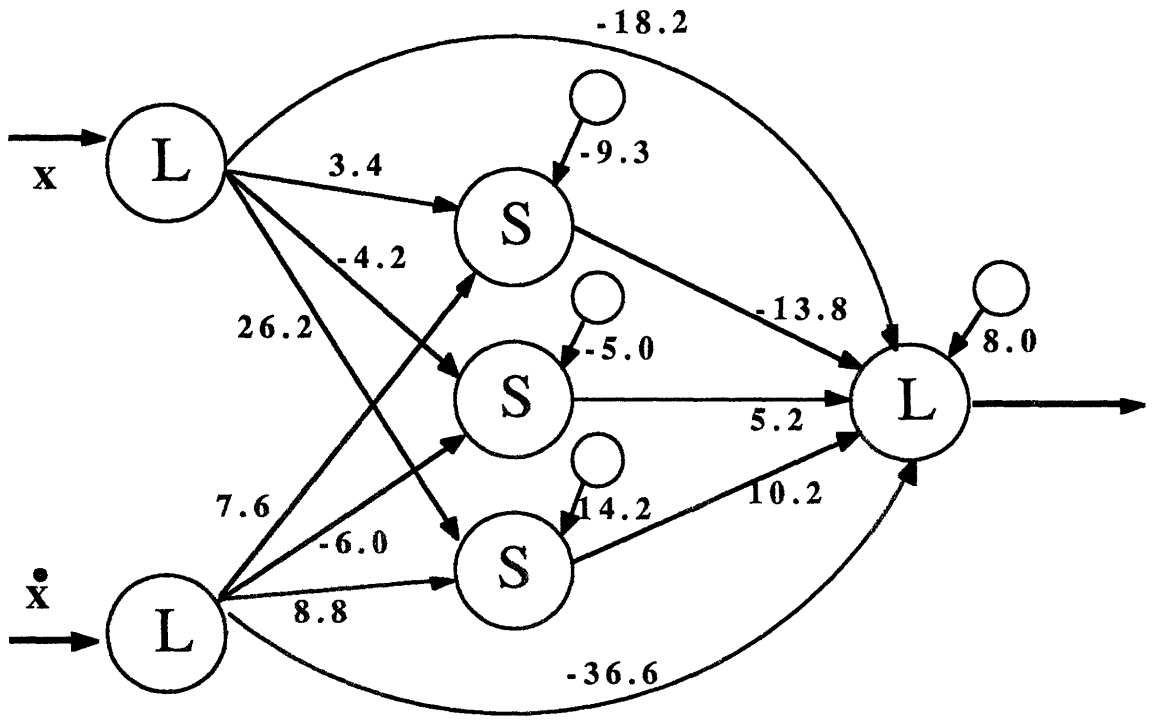

Figure 3.4.2: Network after fifty iterations, model reference trajectory #1

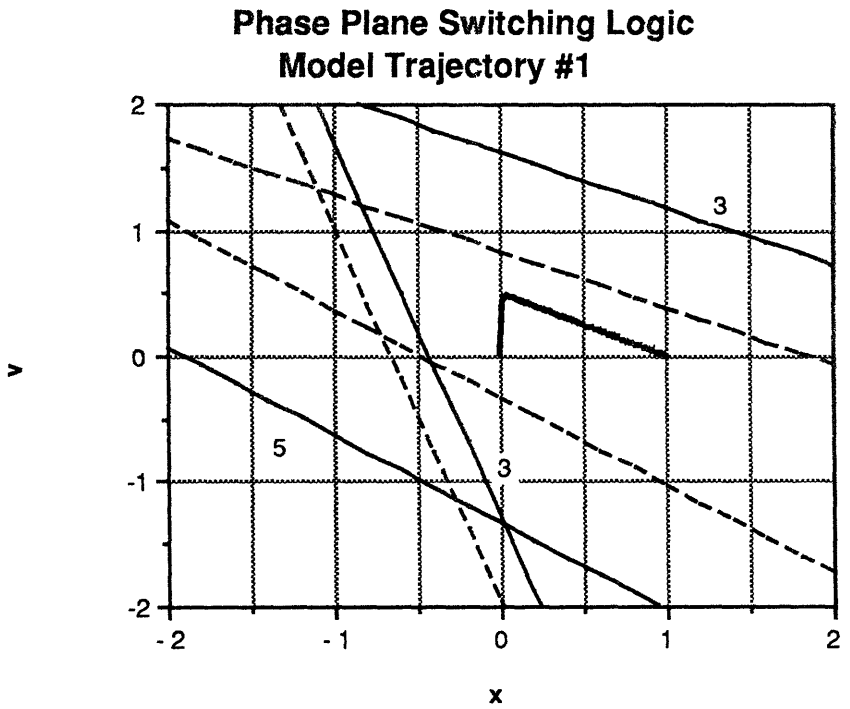

Figure 3.4.3: Switching logic implemented by network of figure 3.4.2

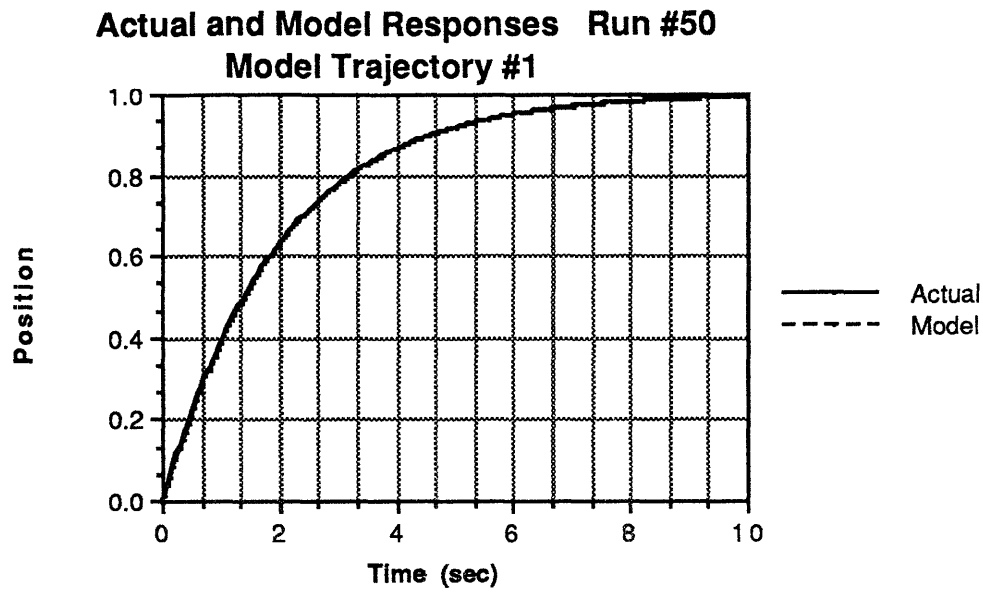

Figure 3.4.4: Comparison of actual and model trajectories for experiment #2

This approximation of the control law as linear state feedback is valid only until the state crosses one or more of the network's switching lines; for example, if neuron number five shuts off due to a different (for example more negative) initial condition on plant position, the above approximation will clearly no longer be valid. Fortunately, the network has used its extra degrees of freedom for this problem to design switching lines which act with the linear portion of the control law to bring the state back into the region where the approximation analyzed above is valid. If the velocity or position grow too strongly positive, neuron number three comes on, introducing -13.8 units of control into the system, pulling it strongly back toward the shaded region. Similarly, if the velocity or position become too strongly negative, neuron five will turn off and neuron four will turn on, introducing a net of between +5 .0 and +10.2 units of control, again forcing the state to the shaded region.

The second experiment had a model position trajectory specified by:

$$x_d(t) = 1 - \exp(-t) \tag{3.20}$$

and, as before, the weights $\mathbf{m}^T = [2.0 \quad 0.0 \quad 0.3]$ were used in the payoff function. Figure 3.4.4 shows the model trajectory and the actual trajectory generated by the network during the fiftieth solo run. There are some slight discrepancies between the two responses, but on the whole the network has done a good job of reproducing the desired response.

Figure 3.4.5 shows the network which implements the network's control law, and Figure 3.4.6 shows the resulting switching lines. Unlike the previous experiment, this control law depends upon neuron number four coming fully on as the network approaches equilibrium, introducing -26.0 units of control and effectively braking the system to a halt. Two features of the control law are especially striking. The first is the very effective velocity limitation system the network has constructed; for velocities greater than about 0.8 (in the vicinity of equilibrium), both neurons three and five simultaneously switch on, introducing a total of -40.7 units of control, strongly reducing the velocity. The trajectory shown proceeds almost tangent to these switching lines but does not actually cross them, suggesting that these neurons are not involved in the response. However, recall that in general the network *does* experience those states, particularly during its first training runs. Based upon those experiences, the network has "learned" that if the state even begins to enter those regions, the plant is deviating substantially from the desired response, and hence the network should slow the plant down.

The second point is that the linear half of the control law uses *positive feedback*! In effect, the network is depending upon the braking action of hidden neuron four to bring the system to a halt; the linear part of the controller could not do this by itself. This is the only experiment conducted for this thesis during which this was observed to happen, and is probably wholly attributable to the fact that the plant was always released from the origin of the state space during the training and solo runs. This is again the specter of the "infinite training set" discussed in Chapter 2; the control laws devised by the network are designed based only on the parts of state space experienced while training. Recall that the previous experiments avoided this pitfall because the negative feedback linear terms dominated for points in state space far from the equilibrium. Here this limitation of the algorithm has not only resulted in a control law which is, in some sense, nonoptimal for certain parts of state space, but is in fact *unstable* for certain combinations of initial conditions. Clearly, for initial position conditions greater than about $x_0 = +2.2$ or less than about $x_0 = -13.2$, the positive position feedback term of the control law will dominate and drive the plant unstable. Once again, a different set of plant initial conditions, enabling the network to experience more of the state space while training, would probably result in a more globally stabilizing controller.

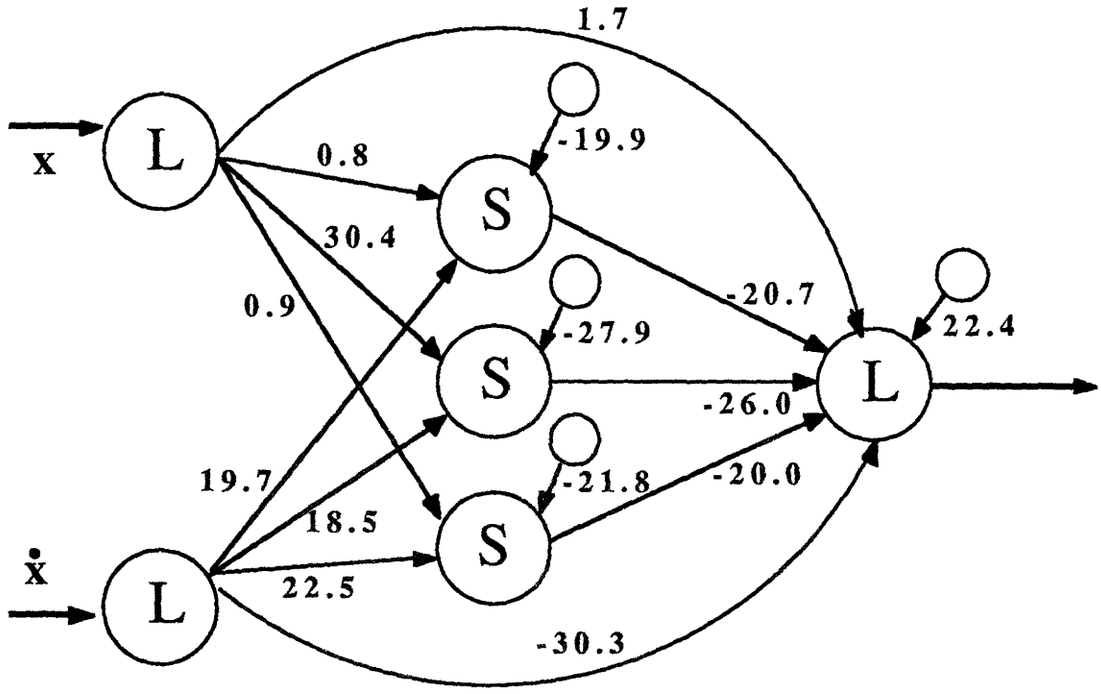

Figure 3.4.5: Network After Fifty Iterations, Model Reference Trajectory #2

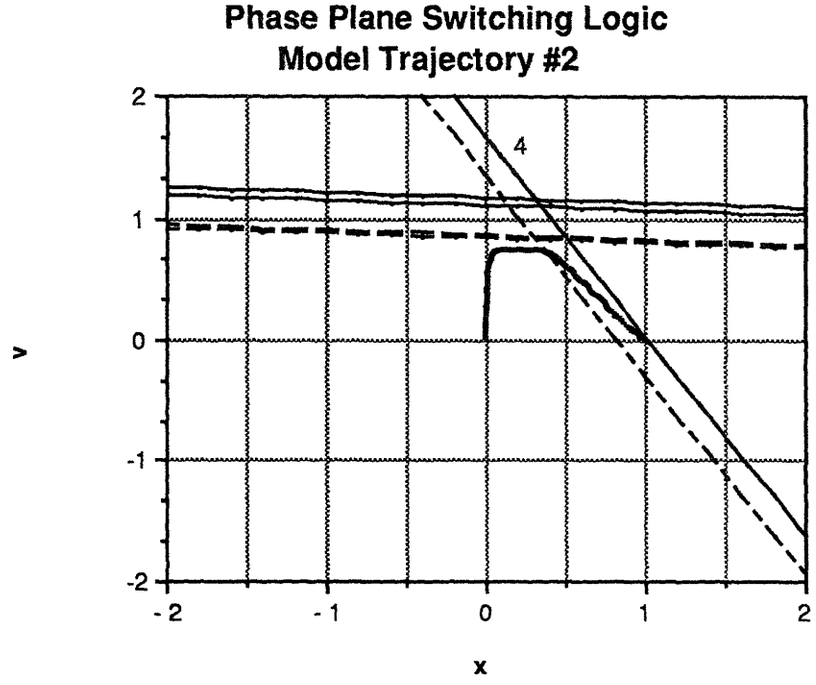

Figure 3.4.6: Switching Logic Implemented by Network of Figure 3.4.5

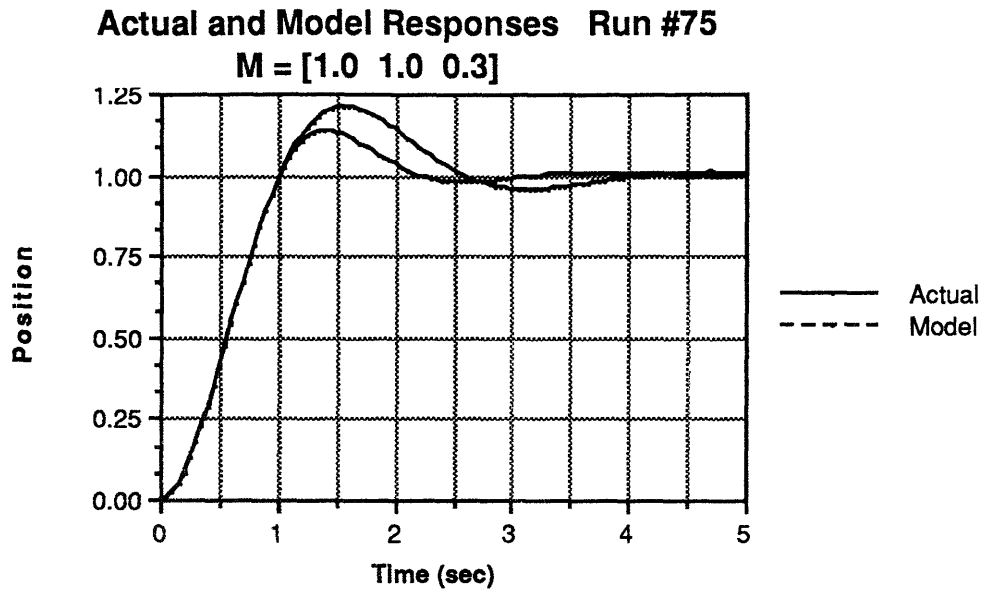

Figure 3.4.7: Comparison of actual and model trajectories for experiment #3

3.4.2 Second Order Trajectory

The last experiment with the model reference payoff function had a model position trajectory specified by:

$$x_d(t) = 1.0 - 1.12\exp(-t)\sin(2t + 1.17) \qquad (3.21)$$

and the model velocity trajectory by:

$$\dot{x}_d = 1.12\exp(-t)\sin(2t + 1.17) - 2.24\exp(-t)\cos(2t + 1.17) \qquad (3.22)$$

The weight vector $\mathbf{m}^T = [1.0 \quad 1.0 \quad 0.3]$ was used. Since the network was observed to still be settling to a constant control law during the fiftieth training run, the simulation was extended to seventy-five runs. Figure 3.4.7 show the model trajectory and the actual trajectory seen in the seventy-fifth solo run. Here it is obvious that the network has not quite devised an adequate solution; the response developed by the network has a more damping and a higher natural frequency than the desired response (although note that the rise times are identical--this is a perfect example of the tradeoff, discussed in the previous section, between damping ratio, natural frequency, and rise time).

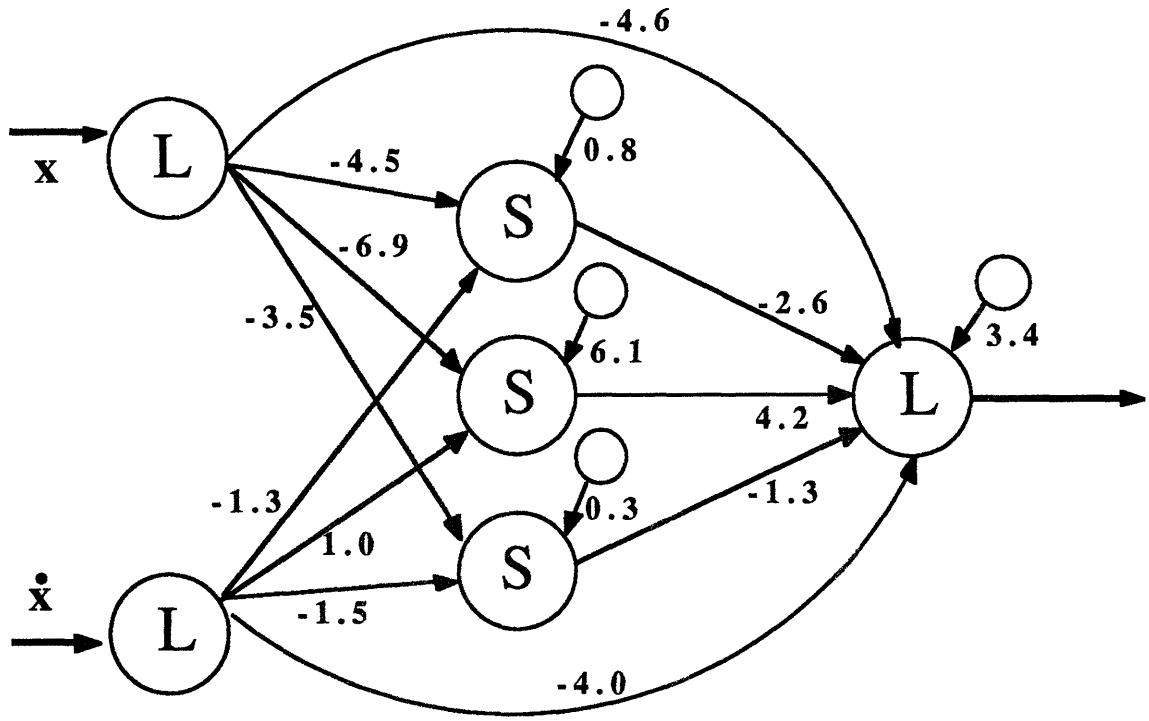

Figure 3.4.8: Network after seventy-five iterations, model reference trajectory #3

Figure 3.4.8 shows the network which developed after the seventy-fifth run and Figure 3.4.9 shows the resulting switching lines. Interestingly, notice that neurons three and five turn off after the state passes through the 0.5 position point, but then turn back on during the first undershoot, contributing negative control action. The network is thus trying to pull the velocity more negative at that point, which does in fact reflect the deviation of the velocity from the model at that point in time, as shown in Figure 3.4.10. Unfortunately, this is too little too late.

The results of this last experiment with the model reference payoff would probably be more satisfying if larger values for M_x and $M_{\dot{x}}$ were used, or if the network were allowed to learn for an even longer period of time. Time constraints, however, prohibited further experimentation in this area.

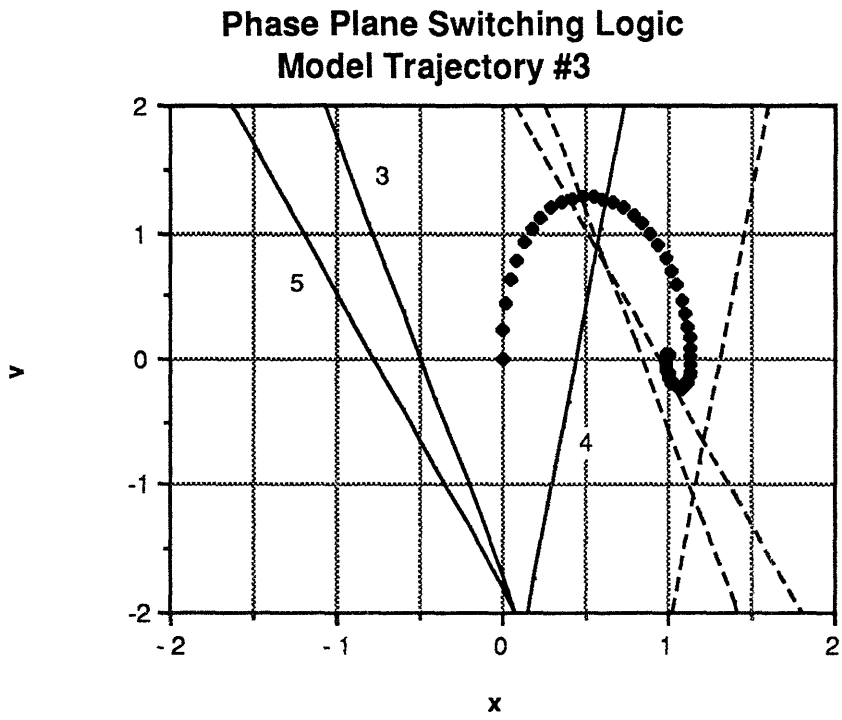

Figure 3.4.9: Switching logic implemented by network of Figure 3.4.8

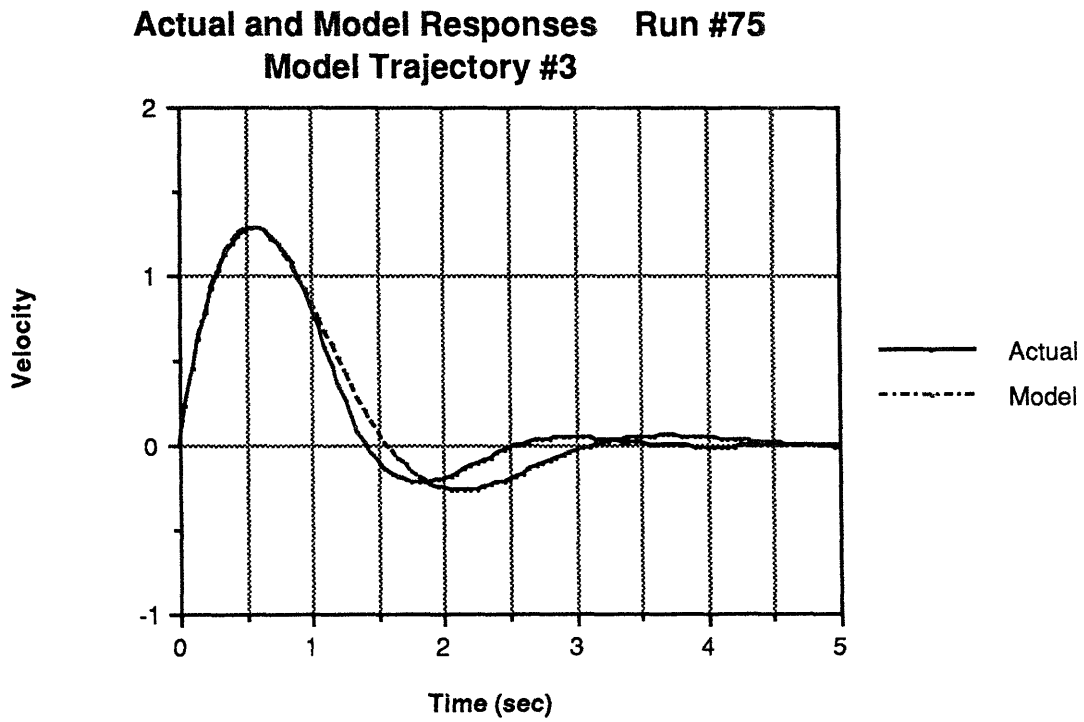

Figure 3.4.10: Comparison of actual and model velocities for experiment #3

3.5 Robustness of the NMC

3.5.1 Changing Plant Dynamics

These experiments sought to determine further the adaptive properties of the NMC algorithm by suddenly changing the properties of the unknown plant after a certain number of training phases. The NMC was permitted to learn to control the double integrator plant for 19 complete runs. Just after the training phase of the 20th run, the dynamics of the plant were changed and this new set of plant dynamics was used for the rest of the simulation. This was not a structural change in the dynamics--the order of the plant remained the same--but rather a change to the location of the plant poles, or to the static sensitivity of the plant. The state weighted payoff function was used for both of the following experiments with $\mathbf{m}^T = [1.0 \quad 0.5 \quad 0.3]$.

For the first experiment, the dynamics were replaced with:

$$G(s) = \frac{1}{10s^2} \tag{3.23}$$

which effectively reduces each of the (linear) feedback gains the network has learned by a factor of 10. Figure 3.5.1 shows the solo responses generated just before and just after the change. The change of plant dynamics has resulted in a closed loop response which is now rather underdamped, with an overshoot of 32%, and which requires almost three times longer to settle. This is just what one would expect if the equivalent linear controller had each of its gains reduced by a factor of 10.

Figure 3.5.2 (a) shows the state of the network just before the plant change and (b) shows the state of the network after thirty retraining phases. Interestingly, the network has not done the "obvious" thing by increasing each of its linear state weights by a factor of ten; in fact there is no clear pattern to the way the weights have been reorganized. The velocity weighting has even been decreased! Figures 3.5.3 and 3.5.4 show the switching lines for before and after the plant change and give a slightly better picture of what is happening. As a result of the change, the network has reduced the width of the linear regions for the hidden neurons and moved the on/off saturation boundaries further to the right of the starting point of the plant. Each hidden neuron now also contributes about 50% more positive control when on. The network has thus accounted for the increased "inertia" of the plant by introducing more positive control when the plant starts from rest, and continuing to apply it for a longer period of time than before the change. As the plant finally

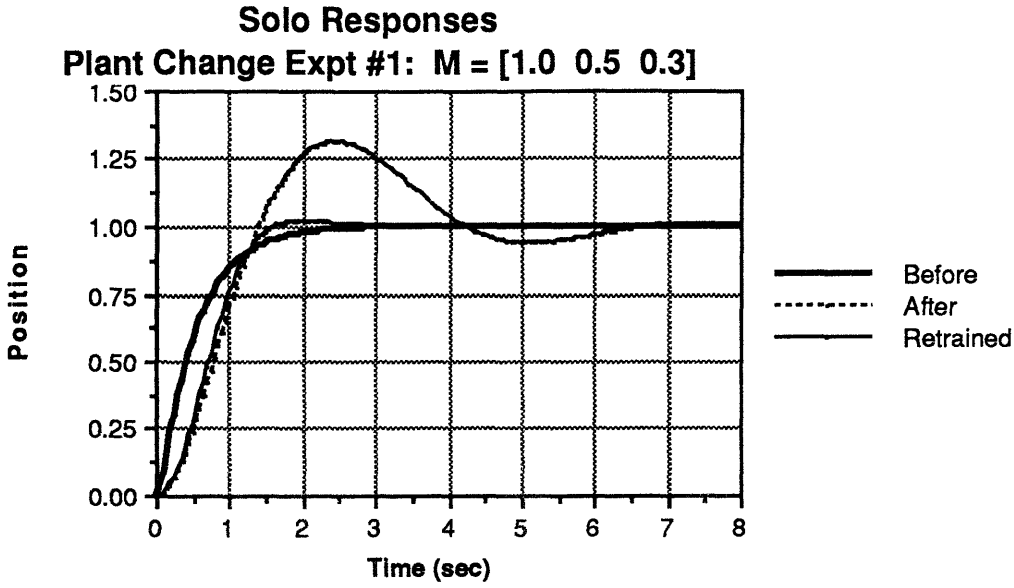

Figure 3.5.1: Solo responses before and after plant change #1, and after retraining

approaches equilibrium, the neurons once again begin operating in their linear region. This new configuration is quite effective in bringing the plant to the desired equilibrium; as can be seen from Figure 3.5.1, the retrained response overshoots by only 2% and has a settling time which is even faster than before the plant was changed.

The results are very similar for the second experiment, in which the double integrator plant is replaced with:

$$G(s) = \frac{3}{(s + 3)(s - 3)} \tag{3.24}$$

Notice from Figure 3.5.5 that this was not actually that dramatic a change, since the feedback gains already learned were enough to ensure stability, but the plant now stabilizes around an incorrect equilibrium position. Indeed, the fact that this was somehow a less difficult problem can be seen by comparing the network configurations before and after the change, Figures 3.5.6 and 3.5.7, and by examining the retrained switching lines shown in Figure 3.5.8 . Except for the small adjustments required to produce equilibrium about the correct position, and to ensure that the correct amount of steady state control (-3 units) is generated, there have been no major changes to the network synaptic strengths. As Figure 3.5.5 demonstrates, after the retraining period the closed loop response is indistinguishable from that obtained before the change.

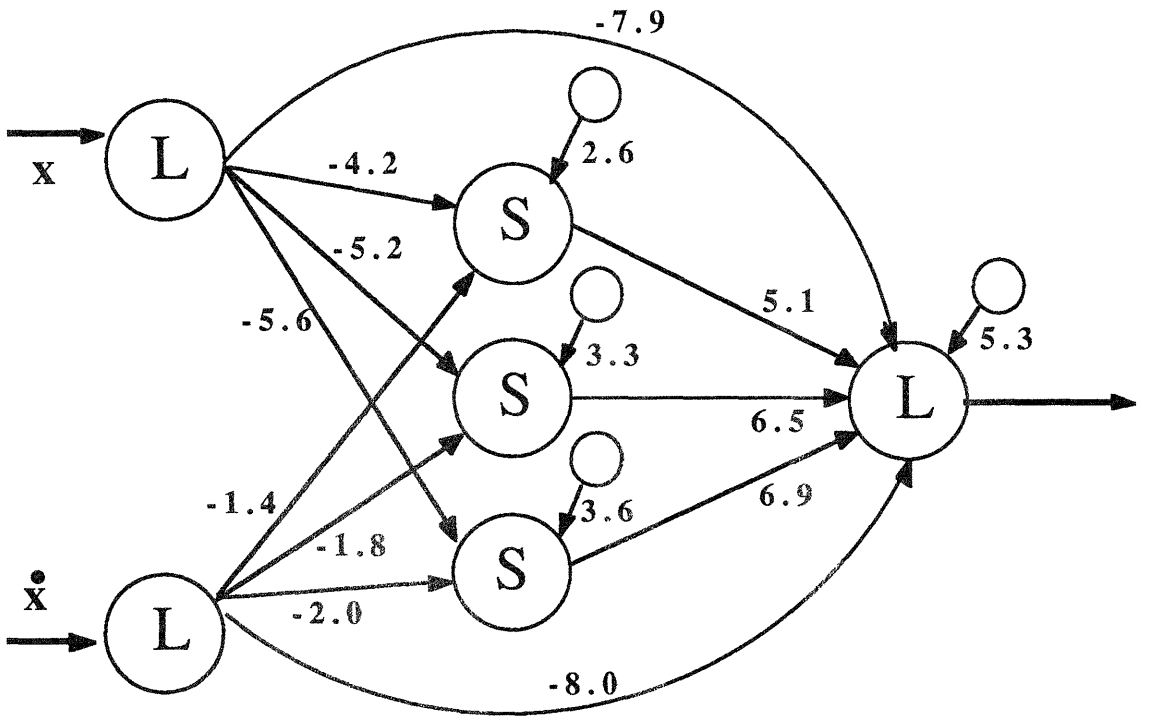

Figure 3.5.2(a): Network before plant change #1

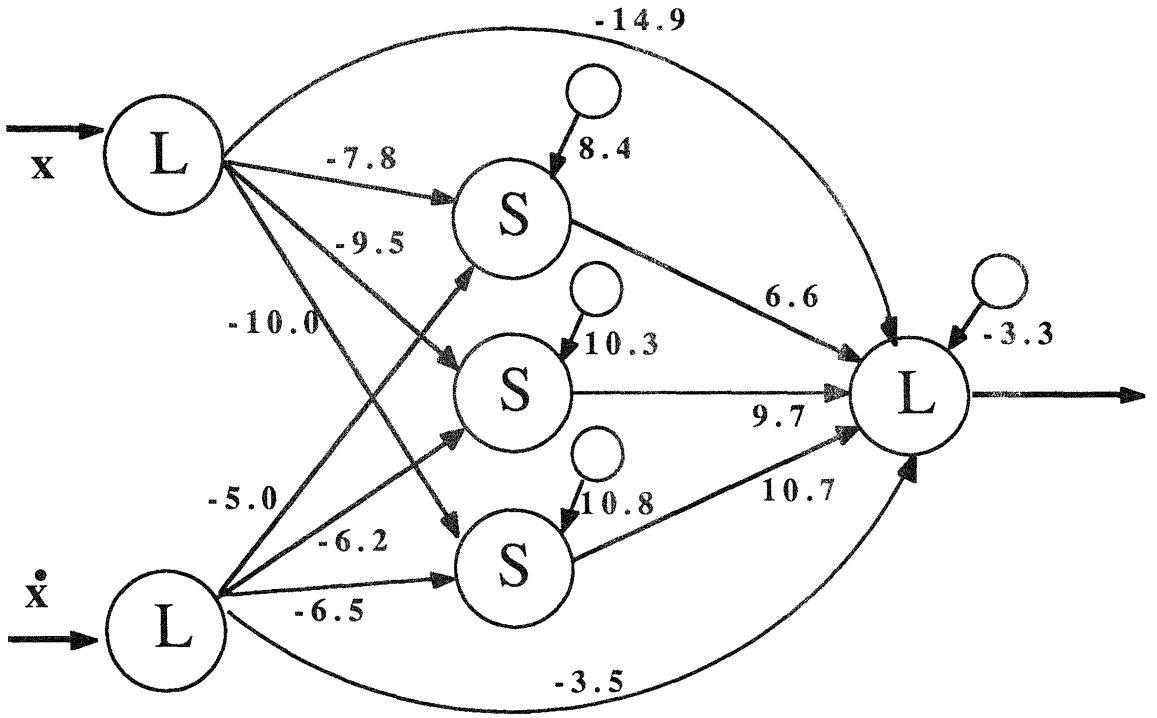

Figure 3.5.2(b): Network after retraining, plant change experiment #1

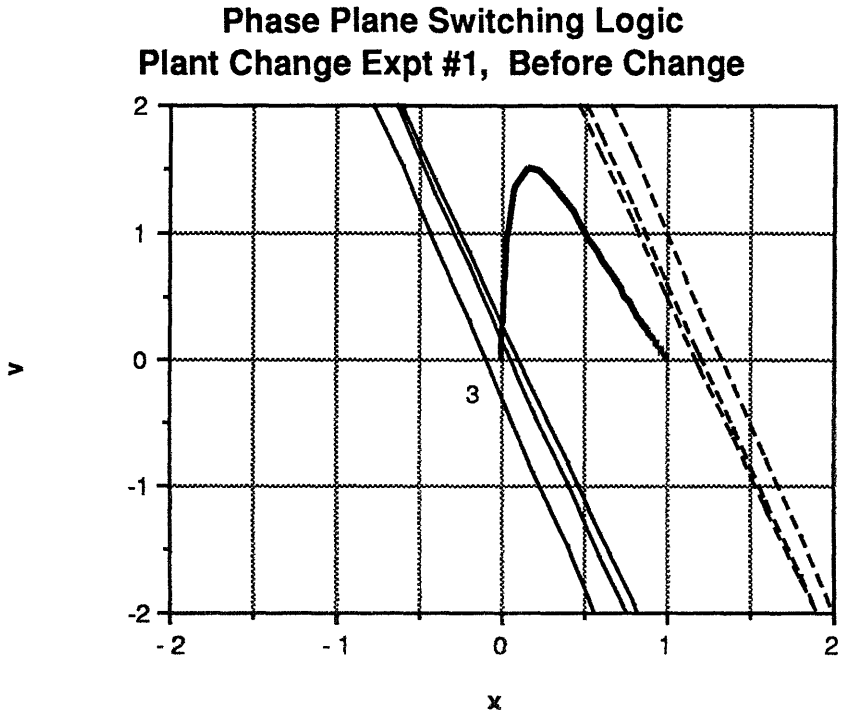

Figure 3.5.3: Switching logic implemented by network before plant change #1

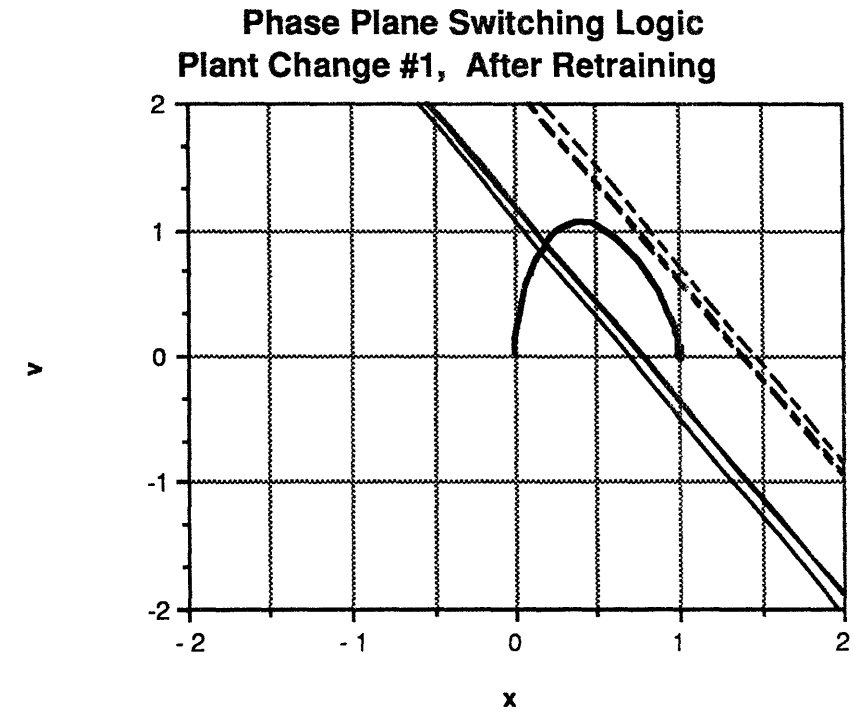

Figure 3.5.4: Switching logic implemented by network after retraining; plant change #1

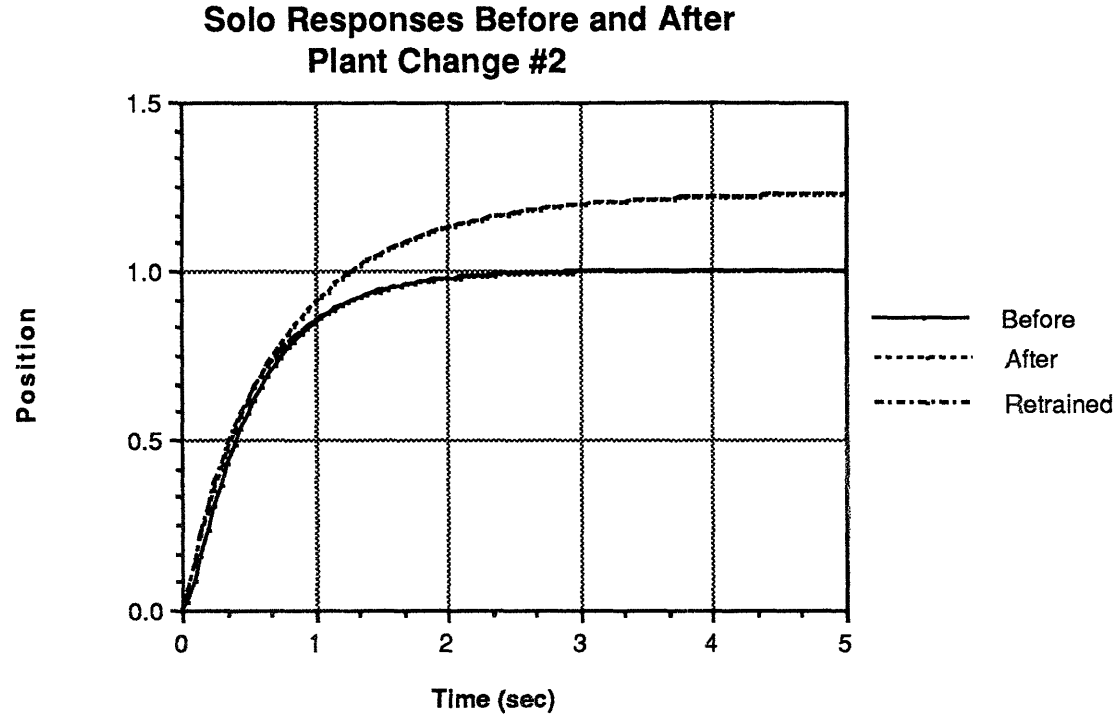

Figure 3.5.5: Solo responses before and after plant change #2, and after retraining

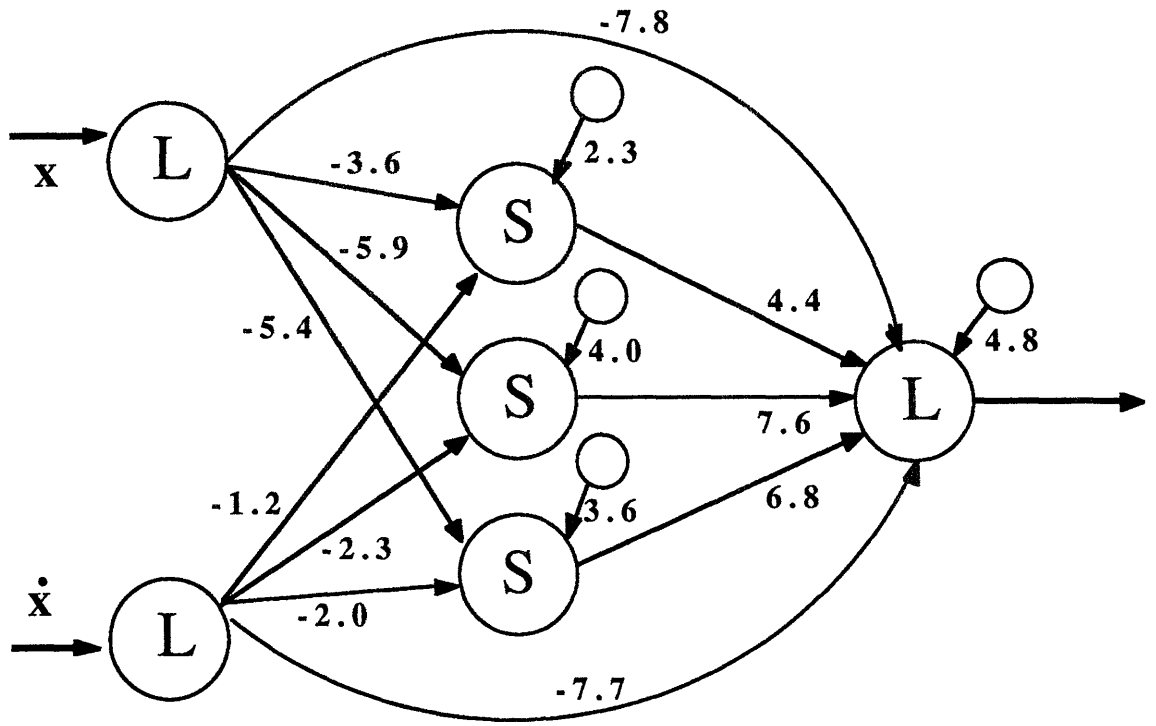

Figure 3.5.6: Network before second plant change

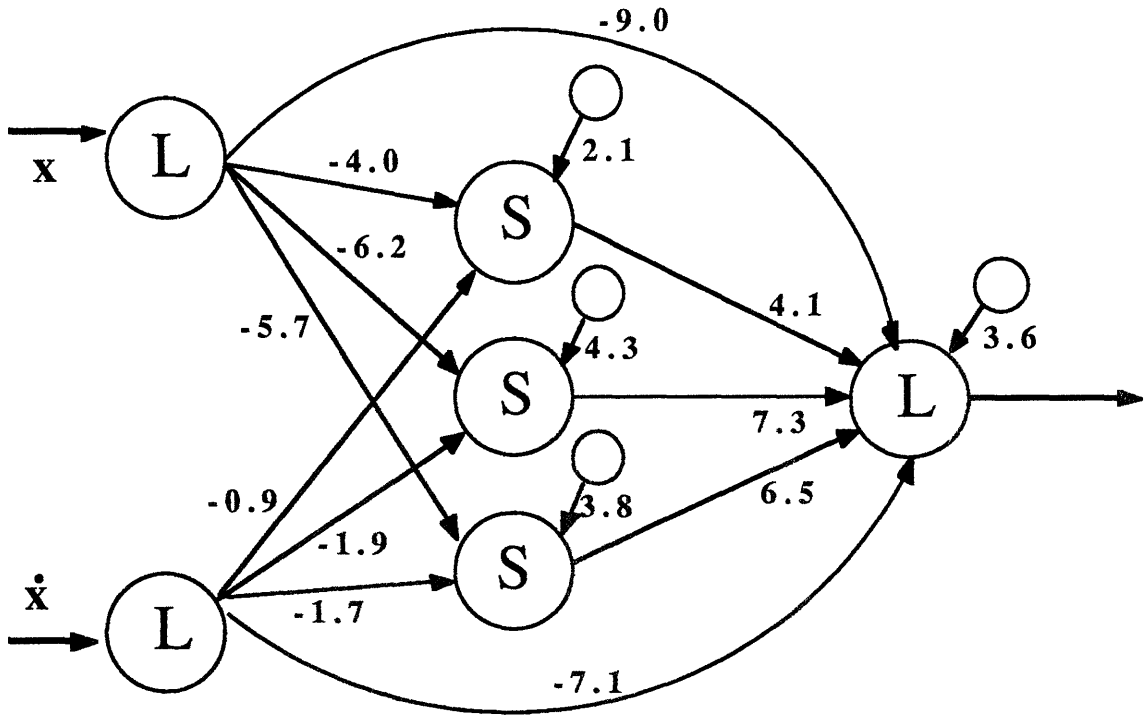

Figure 3.5.7: Network after retraining, for second plant change

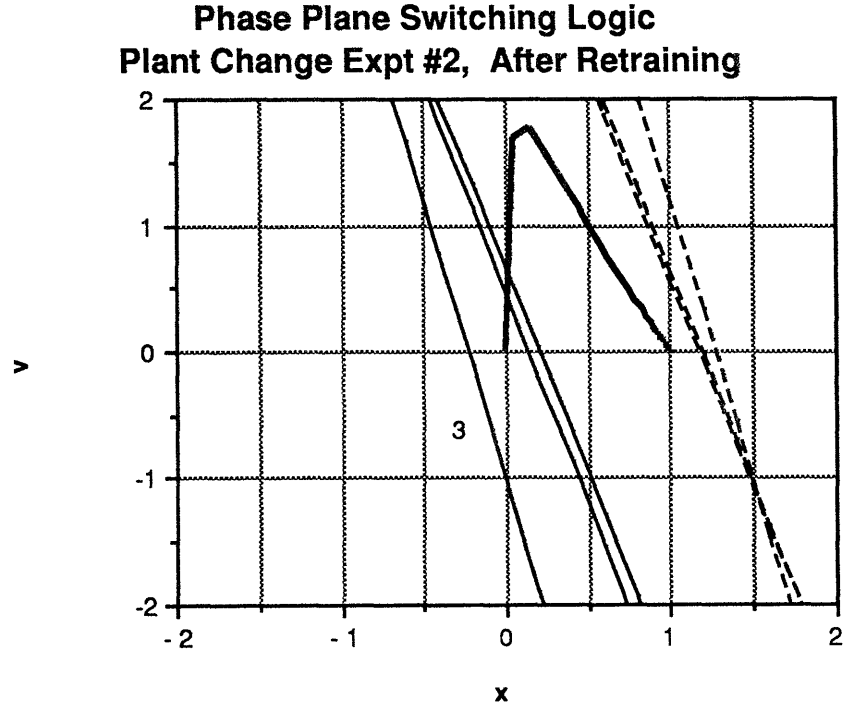

Figure 3.5.8: Switching logic implemented by network after retraining, plant change #2

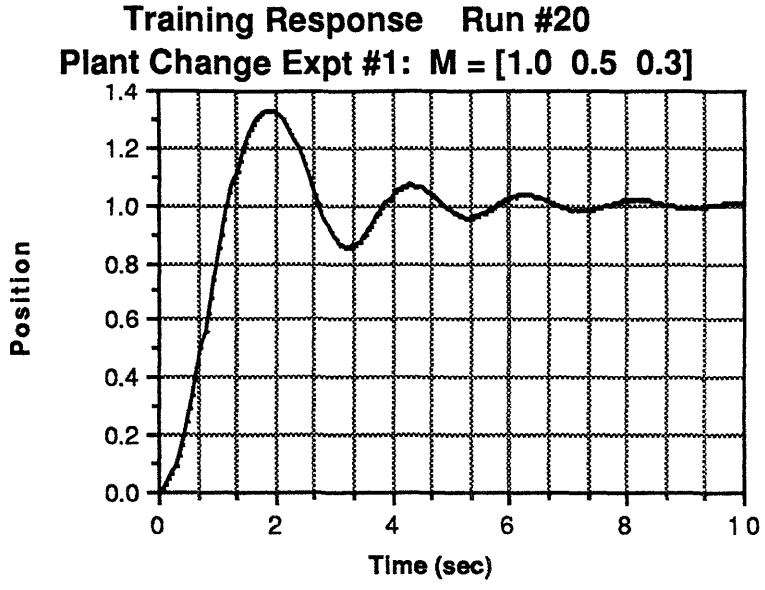

Figure 3.5.9: First retraining phase after plant change experiment #1

As a result of these experiments, one can conclude that not only can the NMC algorithm devise a controller in the absence of *a priori* knowledge of the plant, it can completely revise its controller to accommodate changes in the plant dynamics. As Figures 3.5.9 through 3.5.12 show, adapting to these changes creates transients in the training and solo phases similar to those seen in the original training sequence when the network starts from its initial state of small, random synaptic weights. Eventually, however, the network settles to a steady state control law and the responses seen in each phase and each run become indistinguishable.

3.5.2 Network Synaptic Damage

Above it was noted that damage to the network should result in a graceful degradation of controller performance. To evaluate this explicitly, a network was trained for 19 runs using the state weighted payoff function with $\mathbf{m}^T = [1.0 \quad 0.5 \quad 0.3]$. Before the 19th solo phase, each of the network synaptic weights was then corrupted by a random constant between ±10.0 units. The random numbers were generated using the Microsoft C pseudo random number generator whose statistical properties will be examined in the next section. After the network was thus corrupted, the solo response was generated, then the network was allowed to relearn for five more iterations.

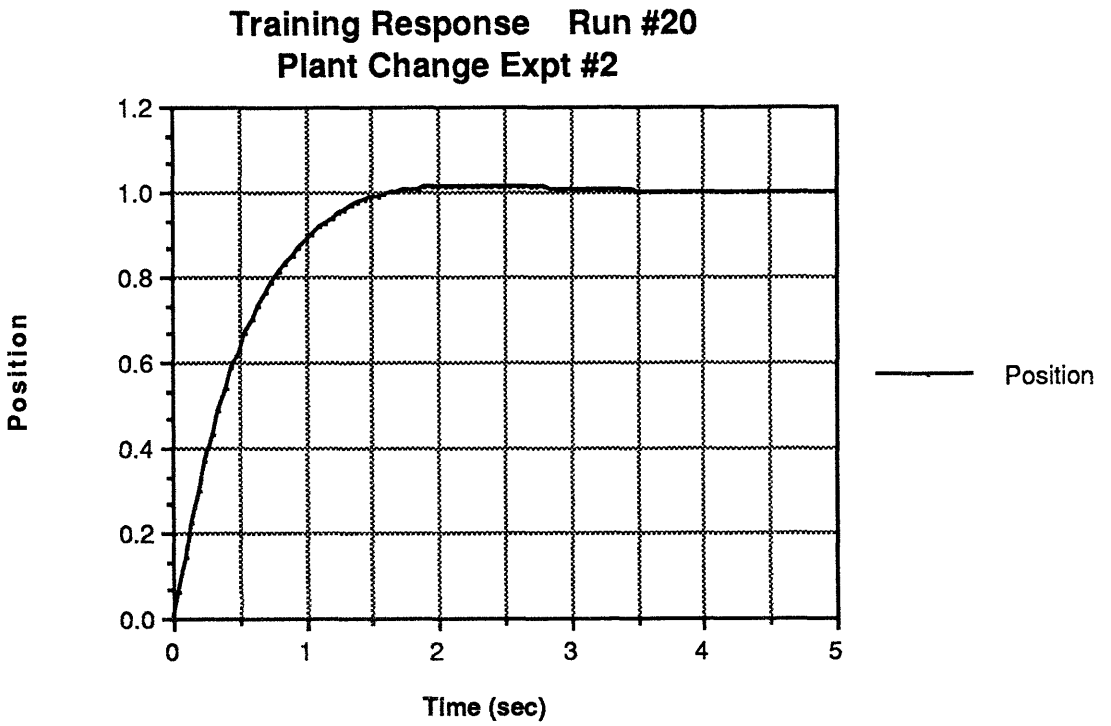

Figure 3.5.10: First retraining phase after plant change experiment #2

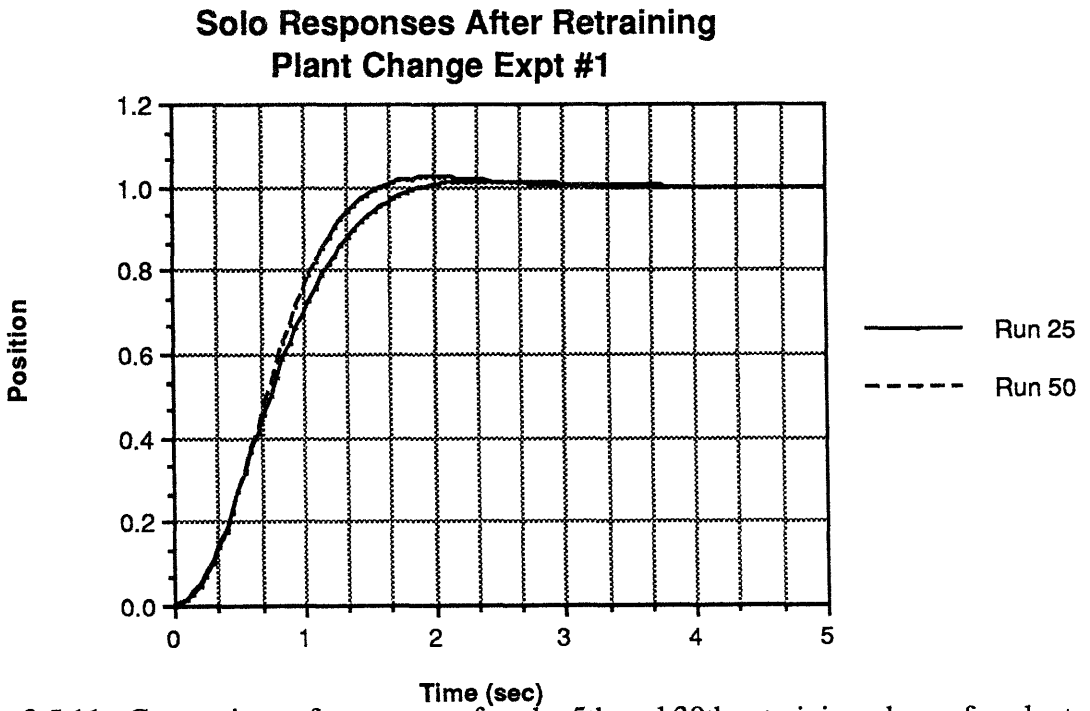

Figure 3.5.11: Comparison of responses after the 5th and 30th retraining phases for plant change experiment #1

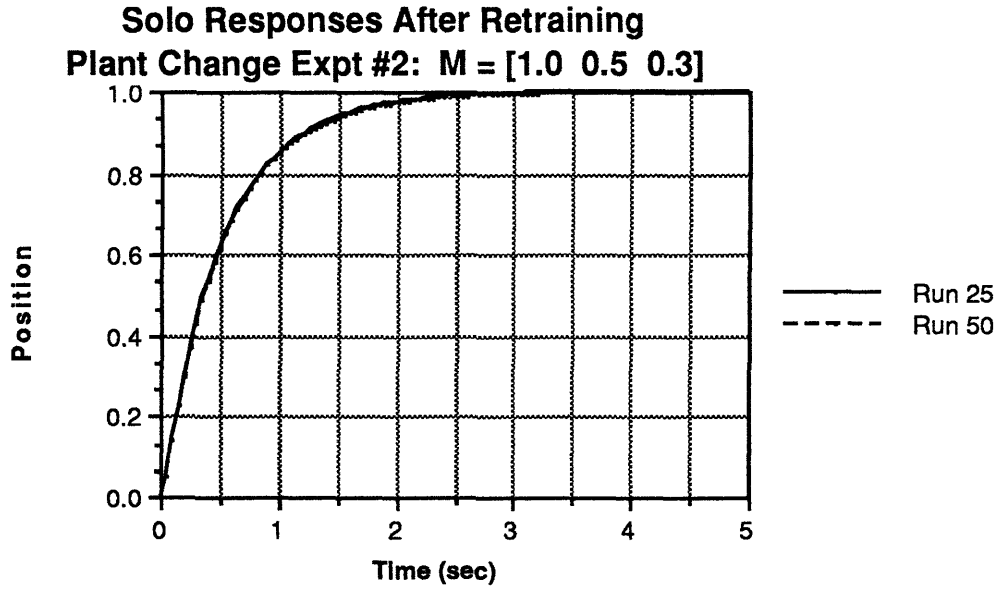

Figure 3.5.12: Comparison of responses after the 5th and 30th retraining phases for plant change experiment #2

Figure 3.5.13 shows the state of the network before (a) and after (b) the weights were scrambled. Note that since the changes did not force either W_4 or W_8 positive, the linear part of the controller is still stabilizing. As Figure 3.5.14 demonstrates, the response is indeed still stable, but around a different equilibrium position. After only five training runs, the response of the retrained network is almost the same as before the damage occurred; the only reason the amount of "recovery" observed in this experiment is not as great as those of the previous section is that this network was allowed only five retraining phases, while those above had thirty.

Figure 3.5.15 demonstrates that, as expected, the initial relearning phase is a bit underdamped, just as though the network were starting again from scratch. However, as Figure 3.5.15 shows, the network has actually had to adjust very little to regain the required equilibrium position, although, as Figures 3.5.16 and 3.5.17 demonstrate, the switching line configuration is very different before and after the damage.

This observation reinforces the conjecture that the switching logic developed by the network is not unique. In fact, this *must* be the case, since from the symmetry of the equations (see Figure 2.10) the hidden neurons are indistinguishable; it is only the initial symmetry breaking randomization of the synaptic weights at startup which allows the development of independent switching lines at all. Hence, switching lines will, in general, be functions not only of the states visited, but also of the initial conditions on the network. In a sense, the control problem being posed to the network is underdetermined; it has

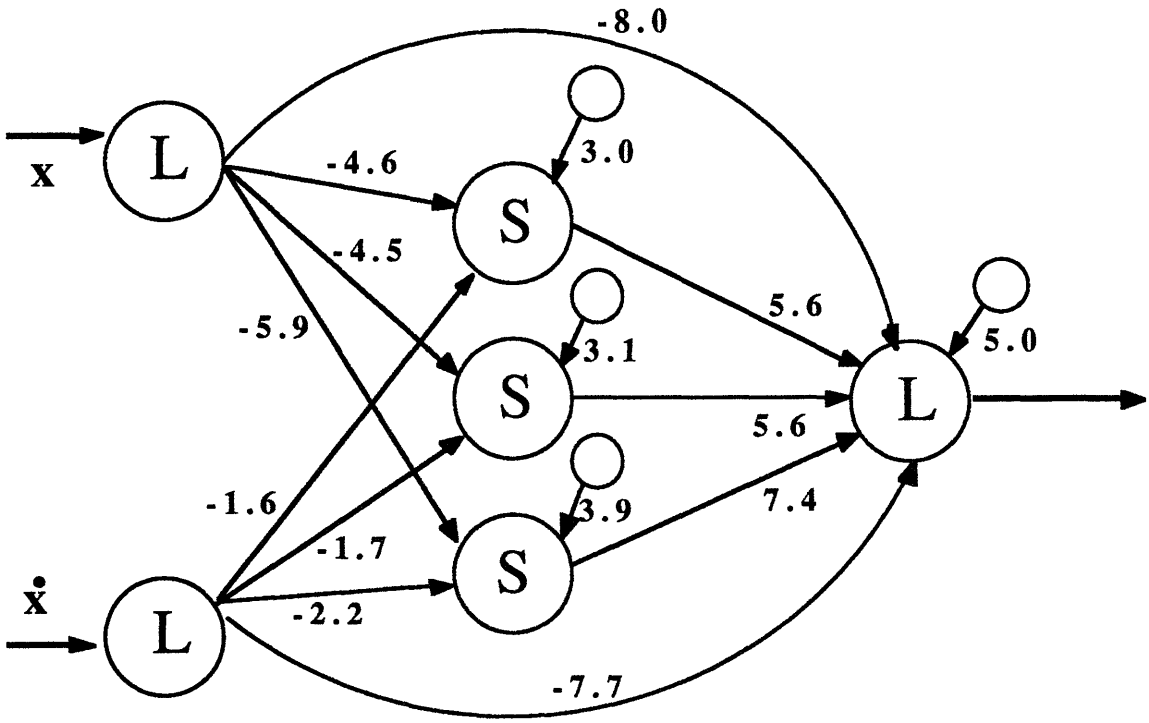

Figure 3.5.13(a): Network configuration before the weights are scrambled

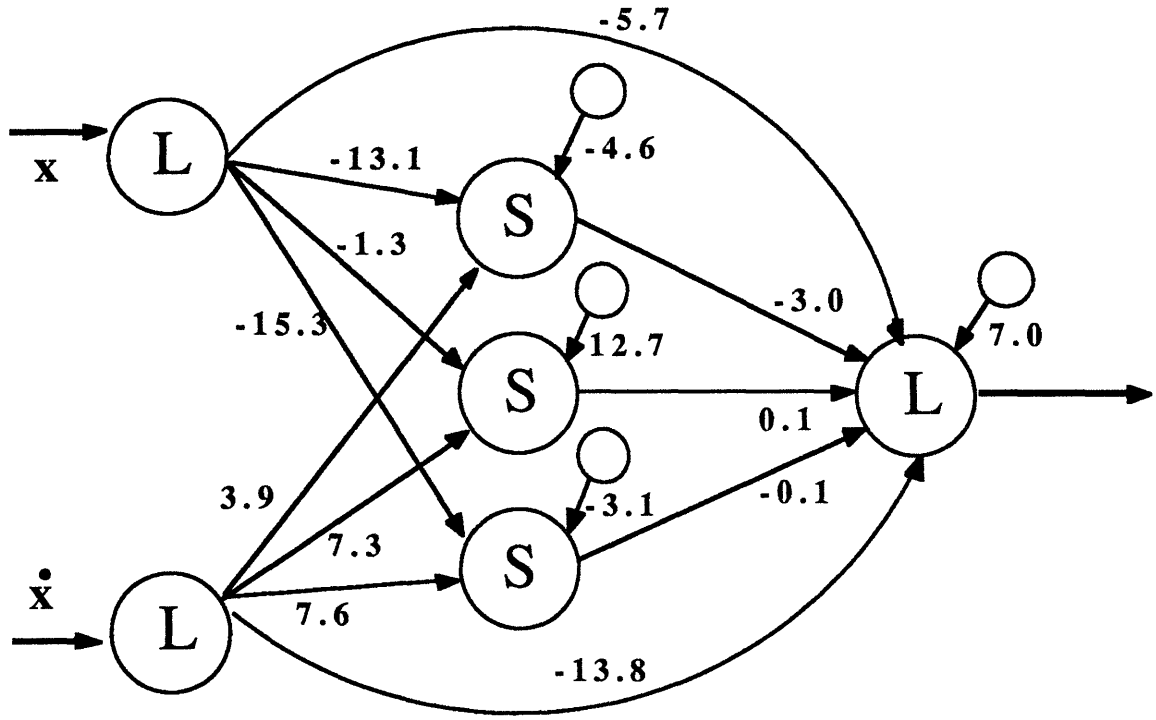

Figure 3.5.13(b): Network configuration after the weights are scrambled

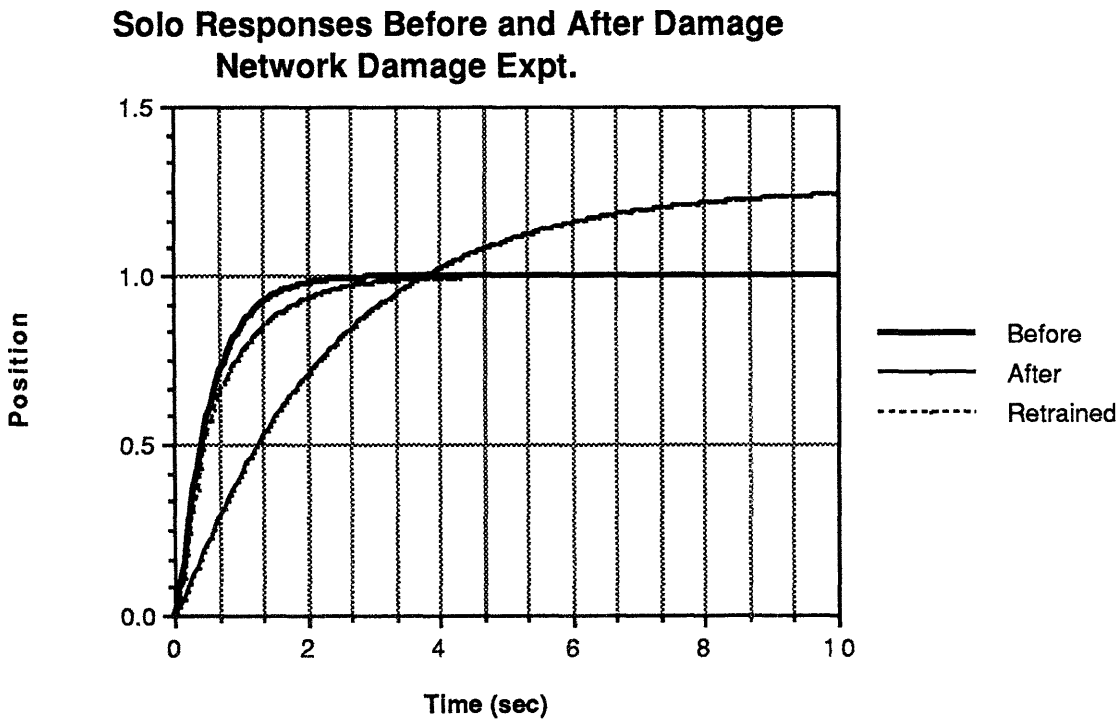

Figure 3.5.14: Responses before and after network damage, and after retraining

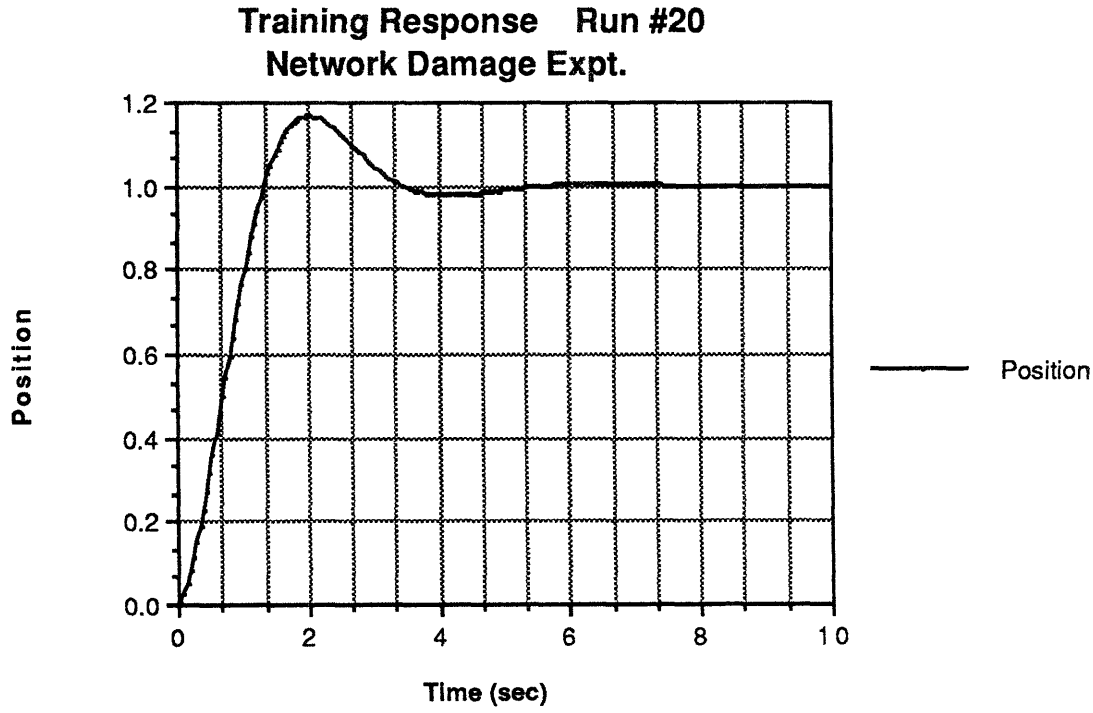

Figure 3.5.15: First retraining phase after network damage

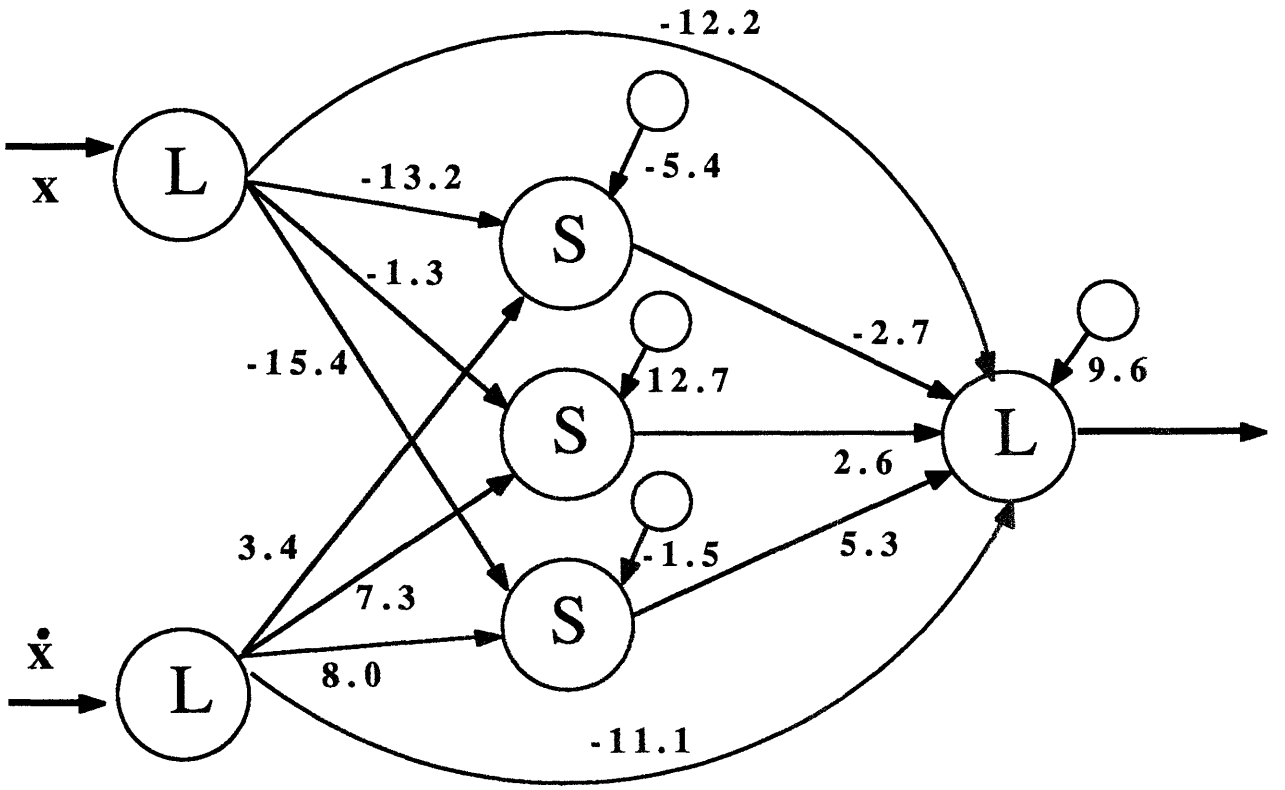

Figure 3.5.16: Retrained network after damage

already been demonstrated that many of the closed loop responses generated using the neuromorphic controller could equally well be generated by a simple linear weighting on the position and velocity neurons plus an additive bias (q.v. Section 2.6). This degree of redundancy suggests that there are many combinations of synaptic weights which will capture the "shape" of the closed loop response implicitly coded in the payoff function. When learning is initiated the network settles to the solution which is closest to the current state of the network weights. More experiments would have to be performed, however, to conclusively illustrate these assertions.

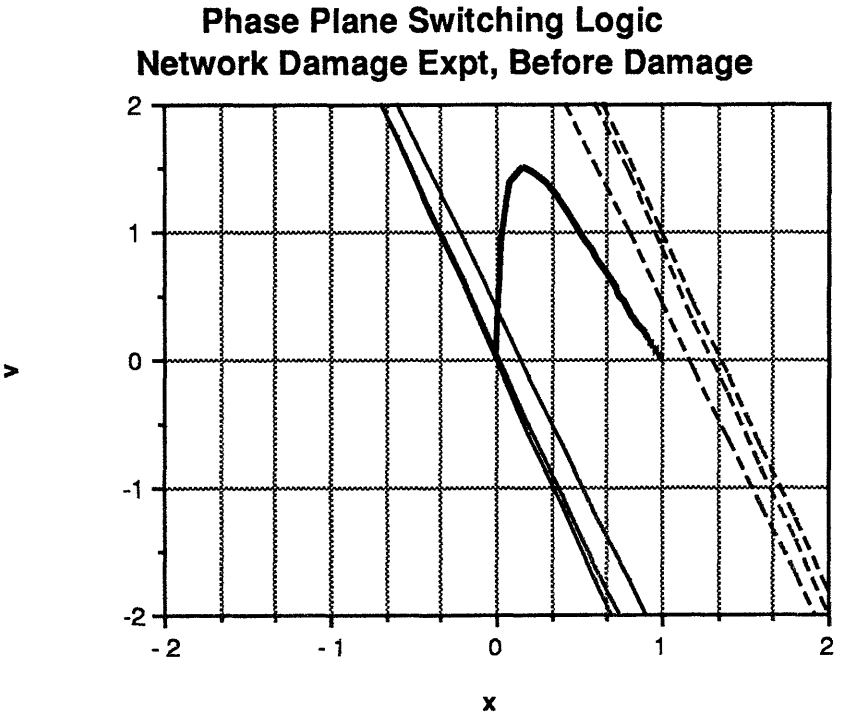

Figure 3.5.17(a): Switching logic before network damage

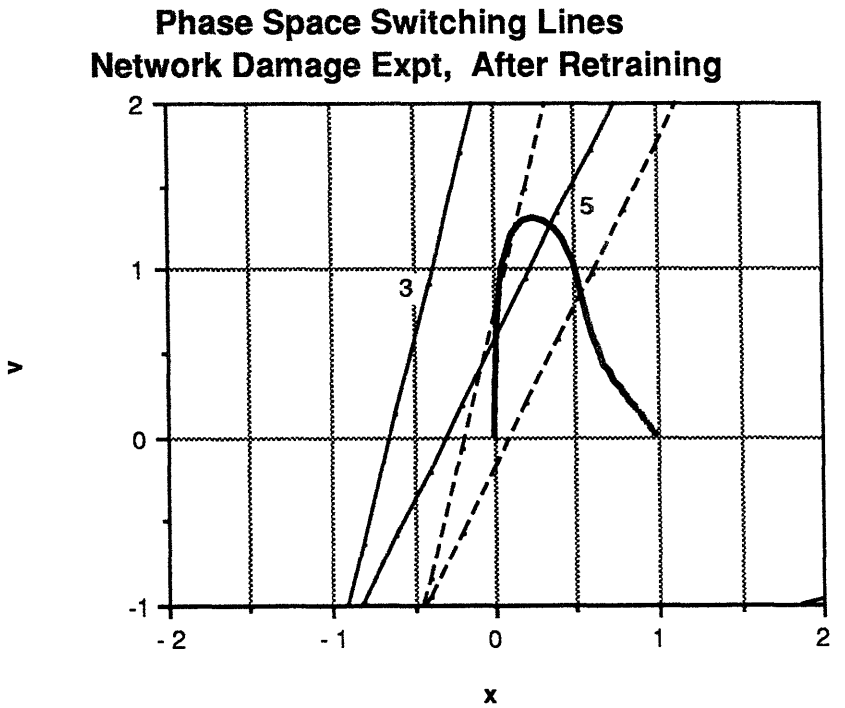

Figure 3.5.17(b): Retrained switching logic after network damage

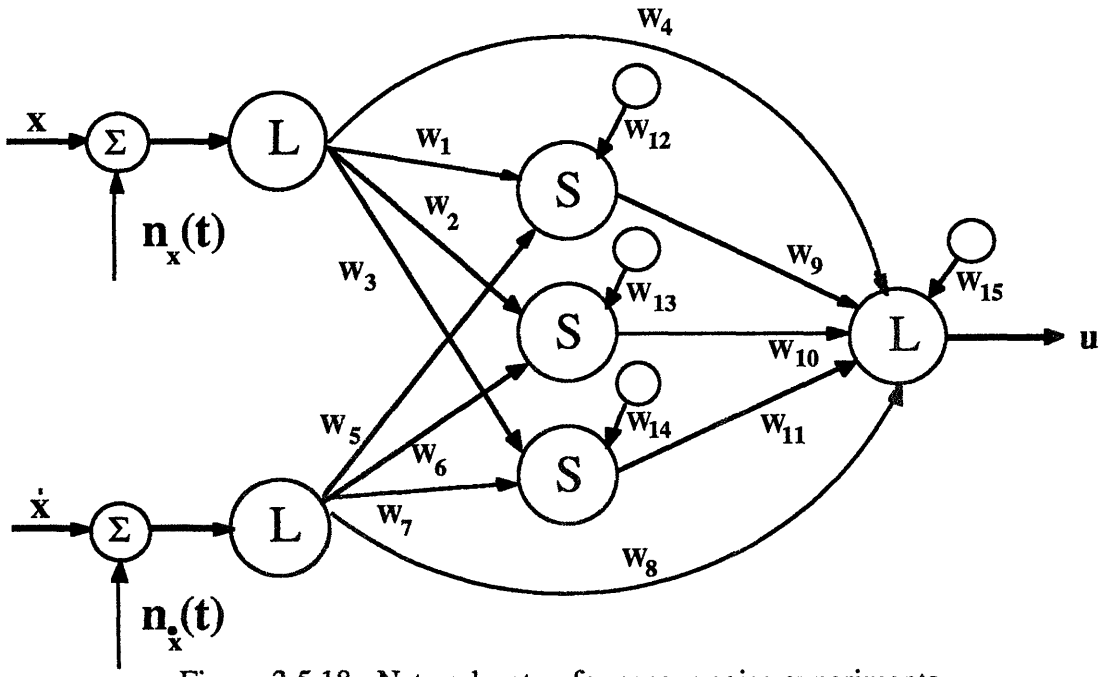

Figure 3.5.18: Network setup for sensor noise experiments

3.5.3 Response to Sensor Noise

The next experiment sought to determine the robustness of the network to additive noise entering at each network input. Figure 3.5.18 shows the block diagram for this experiment. Note that the noise corrupts the values of the state seen by the network, but *not* the values of the state used in computing the payoff function. The network thus never "knows" exactly where the plant is in the state space, but the teacher does. The noise is assumed to be present during both the training and solo phases, and to have constant statistical properties. The important factors to assess in this experiment are: 1.) whether the NMC algorithm can still function in the presence of sensor noise; and 2.) if the algorithm can function, is there any improvement in the solo responses generated in the presence of noise when compared to a network which trained in a noise free environment; i.e., can the network adjust to somehow minimize the impact of the noise in the response? To answer this second question, the solo responses generated by a network which trained in the presence of noise will be compared to those generated by a second network, equal to the first in every way except that it did not train with noise, but which then has noise added during the solo phase. The state weighted payoff function is used for all experiments, with $\mathbf{m}^T = [1.0\ \ 0.5\ \ 0.3]$, and the noise free network used for comparison is the network analyzed in Section 3.3.3.

The noise was implemented by issuing two calls to the Microsoft C (version 5.0) random number generator at each time step, scaling the returned values to the desired

range, and adding the resulting numbers to the current plant state before showing these values to the network. The C pseudo-random number generator returns values which are theoretically uniformly distributed. If this were so, the anticipated statistical properties of the noise would have zero mean and a variance of 0.083 for values in the interval [-.5, .5], and a variance of 0.003 for values in the interval [-.1, .1]. Testing of the actual statistical properties of the function revealed that , for the interval [-.5, .5] the mean is approximately -0.06 with a variance of 0.111, and for the interval [-.1, .1] the mean is about -0.013 with variance 0.004. These statistics were compiled based on five runs of 100,000 trials each for both of the indicated ranges. The computed variances and means varied negligibly during each of the runs, so it is a fairly safe conclusion that the statistical properties of the random number generator are stationary. Notice, thus, that the noise added to the network's input neurons has a nonzero bias in addition to its other statistical properties.

Figure 3.5.19 shows the results of this experiment for a noise level of [-.1, .1]. The top curve is the solo response on the 50th run of a network which did not train with noise, but then had the noise added; the influence of the sensor biases are clearly shown in this response. The bottom curve shows the 50th response of the network which did train with the noise; notice the bias has been eliminated in the response. Figure 3.5.20 shows the same results even more dramatically for a noise level of [-.5, .5]. The steady state statistical properties of these responses confirm what the figures reveal qualitatively. The top curve of Figure 3.5.19 has a steady state mean of 1.026 and a standard deviation of 0.008; the bottom curve has mean of 0.995 and standard deviation of 0.004. The top curve of Figure 3.5.20 has a steady state mean of 1.242 and a standard deviation of 0.023; the bottom curve has mean of 0.993 and standard deviation of 0.019. It would appear that not only does the NMC algorithm for this system function in the presence of noise, it produces networks which exhibit better noise rejection properties than those which train in noise free environments.

What is interesting to investigate is exactly how the network has accomplished this improvement. The task of learning the bias is not difficult, since the bias could easily be considered a property of the (unknown) plant itself, and it has already been shown that the network is quite capable of adjusting to these properties. The real issue is how the network has used its ability to design nonlinear switching lines. Compare Figure 3.5.17, the switching lines of the network which did not train in the presence of noise, with Figures 3.5.21 and 3.5.22 which show the switching lines developed by the networks which did train with noise. The gray squares underlying the region around the equilibrium point indicate the uncertainty in the network's knowledge of the state; due to the noise, when the plant is (actually) at the desired equilibrium, it will look to the network as though it is

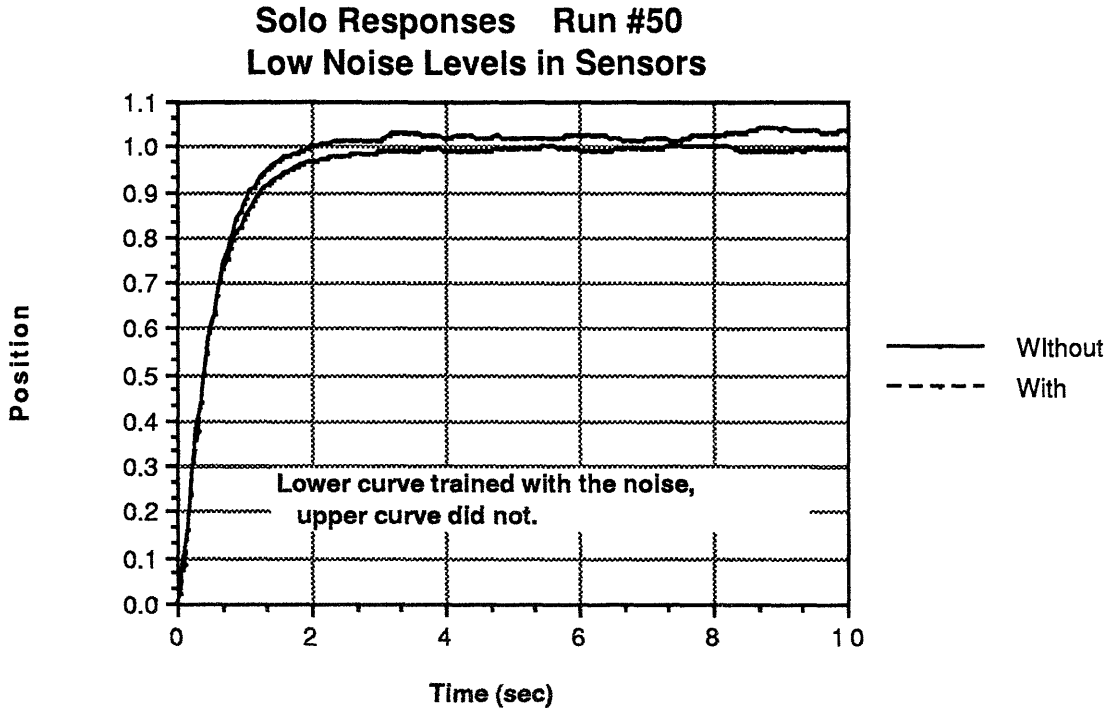

Figure 3.5.19: Solo responses for networks which trained without and with low noise levels

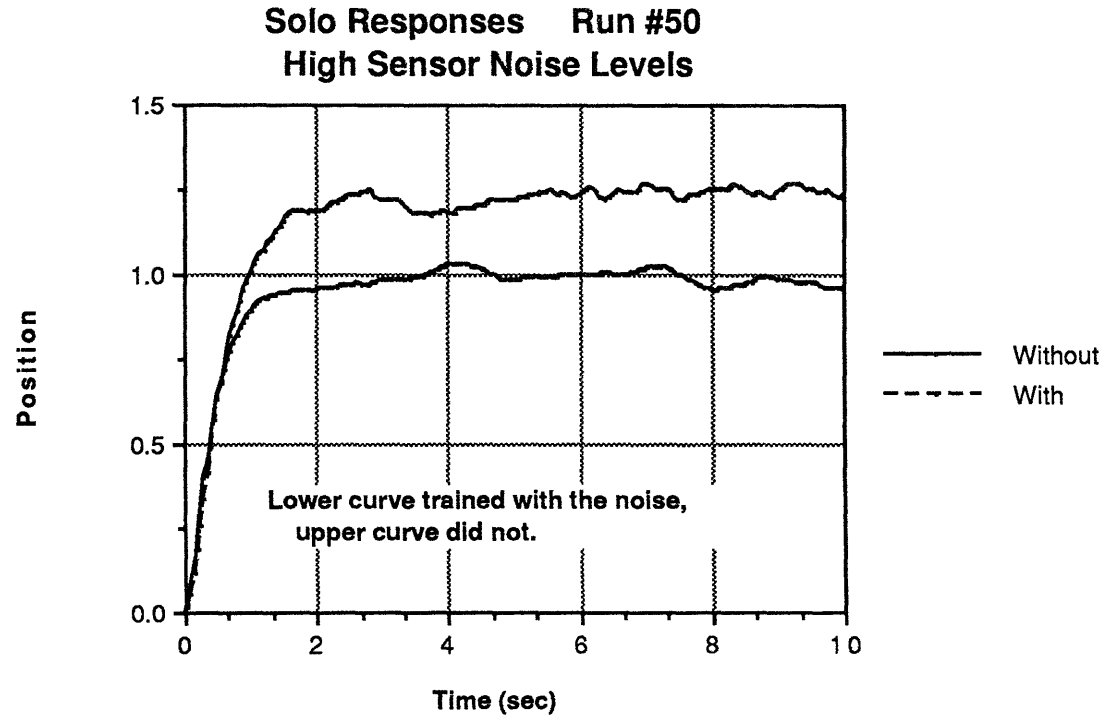

Figure 3.5.20: Solo responses for networks trained with and without high noise levels

scattered throughout the gray area. Notice that the effect of the noise has been to push the "off" saturation boundaries for all the hidden neurons to the left, away from the equilibrium point; the "on" saturation boundaries have been similarly shifted, but not by as much. The equilibrium point now lies in the off saturation region of (almost) all the hidden neurons, while the noise free switching lines have the equilibrium point almost in the middle of the linear region of the hidden neurons.

Is this a solution which makes sense? Recall from Section 3.2 that when neurons are operating in their linear region, the response of the system is the same as a linear feedback system with a higher bandwidth than would be expected from just the direct linear connections of the network (weights four and eight). If such a network were subjected to noise, it would response very strongly since both the linear and nonlinear parts of the controller would contribute. A good solution for the network in the presence of noise would be to design the switching lines in such a way that the nonlinear elements operate most of the time in saturation; completely on for values of the state far away from the equilibrium position, and completely off for state values close to equilibrium. In this way, the only part of the controller responding to noise when the plant is near the desired equilibrium would be the relatively low bandwidth linear term. The controller could thus implement a response which settles to equilibrium like a high bandwidth system, but which responds to sensor noise with much lower effective feedback gains.

To accomplish this the network would need to move the on and off saturation lines closer together to reduce the size of the linear region, and also move the off saturation lines far enough to the left of the equilibrium point that, when the state is near equilibrium, the hidden neurons will not come on in response to the noisy sensor readings. This last requirement is equivalent to ensuring that the off saturation boundaries of each hidden neuron lie outside of the gray boxes.

Looking at Figures 3.5.21 through 3.5.24, it appears that this is just what the network has tried to do. In Figure 3.5.21, while the widths of the linear regions are approximately the same as in Figure 3.3.17 (one is even a bit larger!), the equilibrium point is clearly to the left of the off saturation boundaries of two of the hidden neurons, and these switching lines are just at the lower left edge of the uncertainty region. Although the equilibrium state is still in the linear region of hidden neuron number three, note that this is the neuron which contributes the least to the nonlinear term of the control law, almost by a factor of two compared to the other hidden neurons (+3.6 when on, versus +7.7 for neuron four, and +6.7 for neuron five). The effect is even more pronounced in Figure 3.5.22. The widths of the linear regions are certainly reduced, and all three of the off saturation boundaries lie in the lower left-hand corner of the uncertainty region.

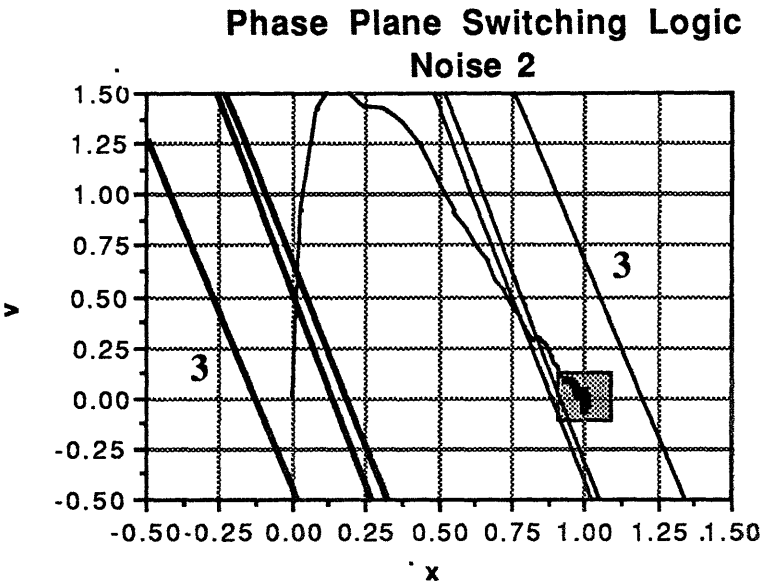

Figure 3.5.21: Switching lines for low noise network

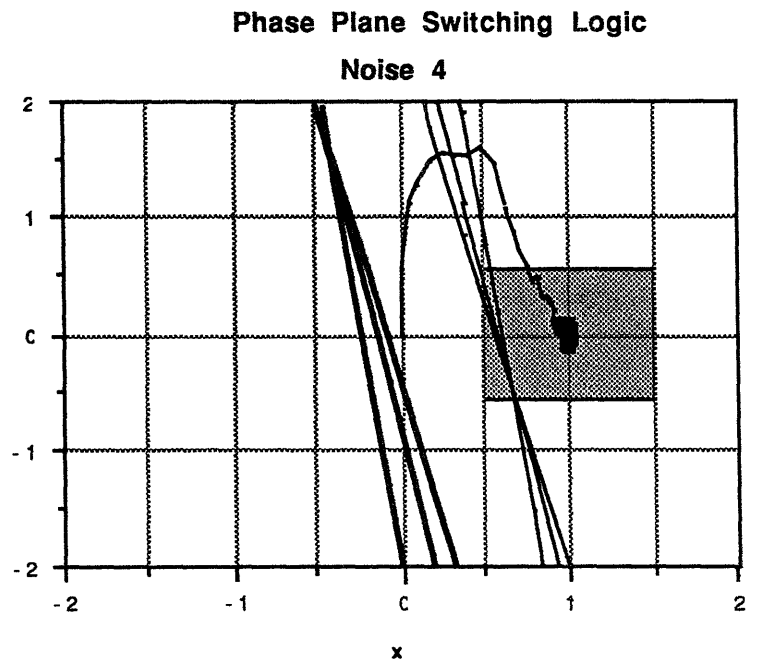

Figure 3.5.22: Switching lines for high noise network

The solutions devised by NMC to the noise problem do thus indeed make intuitive sense. The algorithm has used its degrees of freedom to create nonlinear switching boundaries which bring the plant to equilibrium with a bandwidth higher than that which characterizes its response to sensor noise. There is not necessarily anything "optimal" about this solution, but it is encouraging that the algorithm can not only tolerate noise, but in fact react to the noise in an intuitively correct manner.

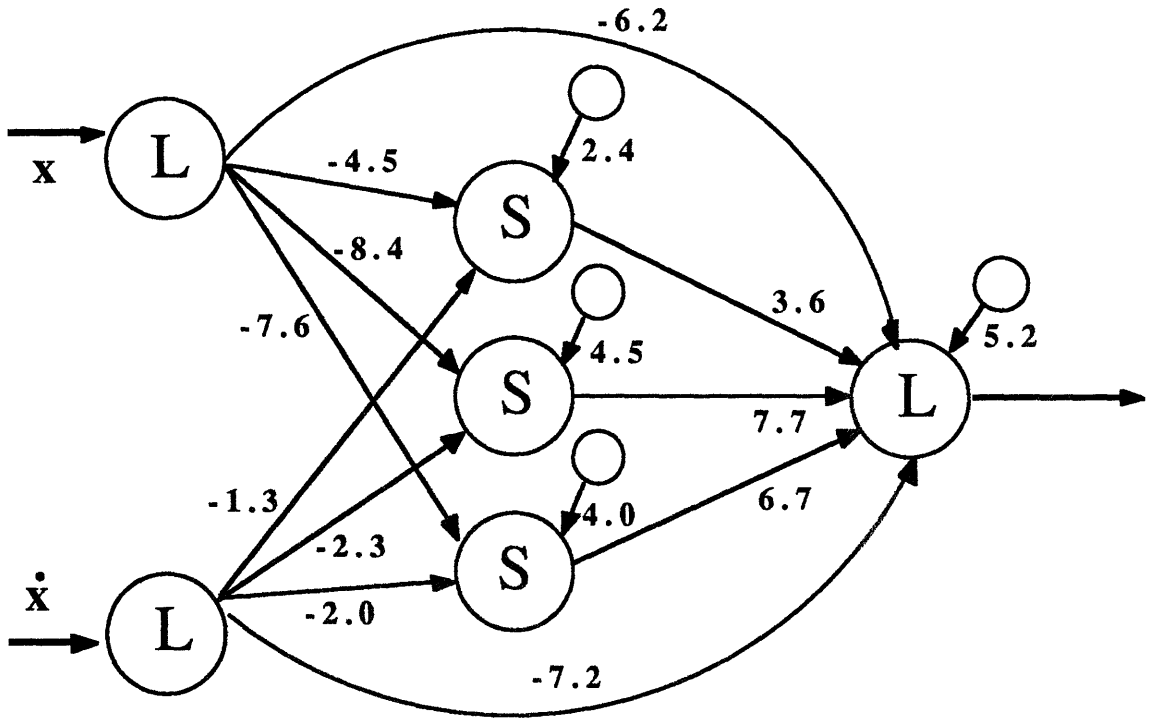

Figure 3.5.23: Network which implements the switching lines of Figure 3.5.21

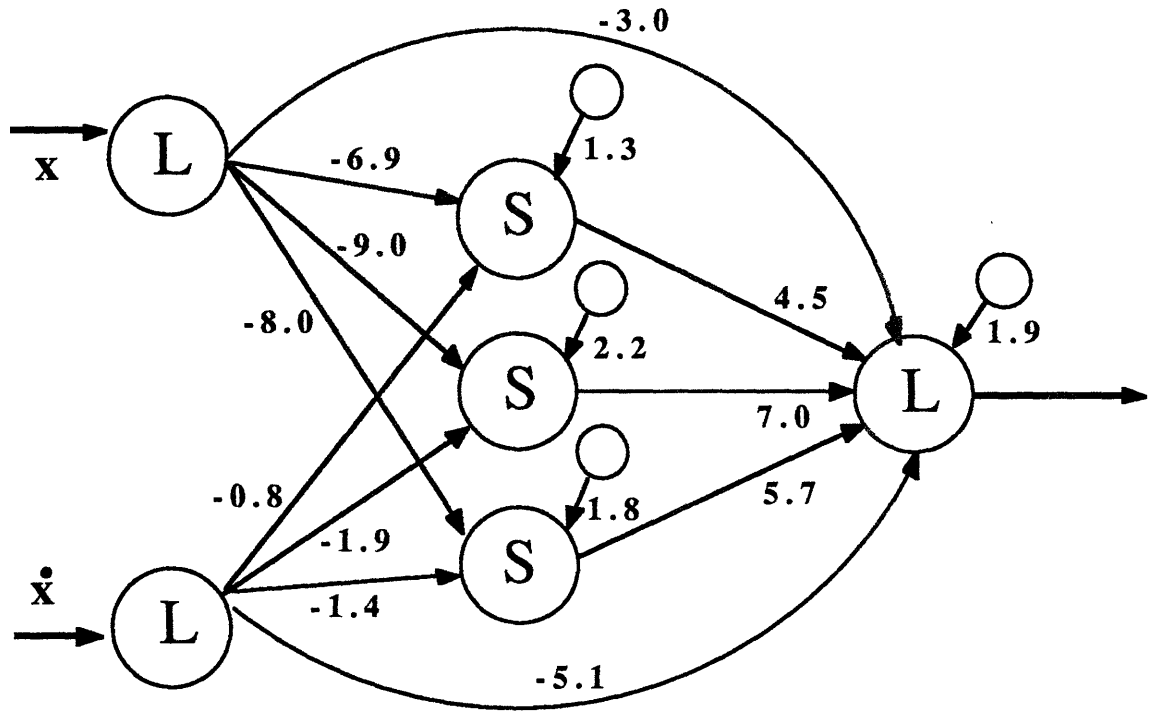

Figure 3.5.24: Network which implements the switching lines of Figure 3.5.22

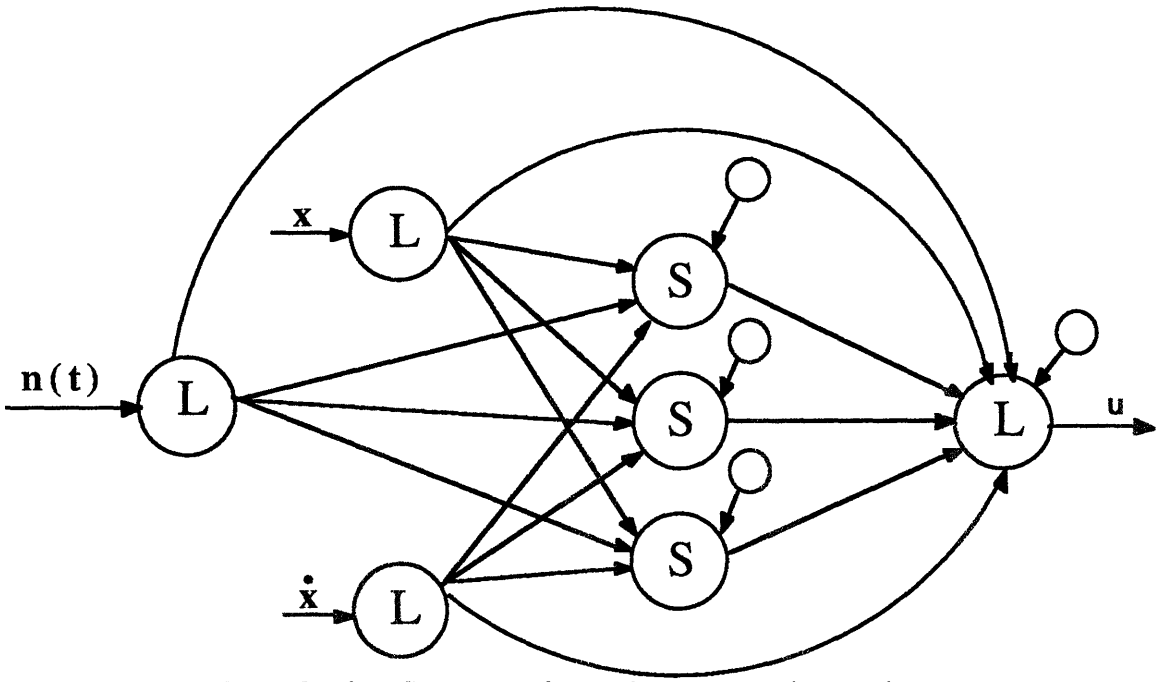

Figure 3.5.25: Structure of extra input network experiment

3.5.4 Response to Irrelevant Inputs

As a final test of the robustness of the NMC algorithm, an experiment was designed to evaluate the effects of a change in the input structure of the network. Figure 3.5.25 shows the structure of the new network; an extra linear response neuron was added to the input layer of the network and given axonic connections to every neuron in both the hidden and output layers, as shown in the figure. This neuron was then driven by the [-.5, .5] noise used in the previous section. The output of this neuron was thus stochastic with a mean of -0.06 and a standard deviation of 0.111 units.

Since this neuron carries no information which would be "useful" to the network, in the sense of helping it minimize the magnitude of the payoff function, the optimal solution would be for the network to assign zero weights to all the synaptic connections made by this neuron. In fact, this is almost what occurs. Figure 3.5.26 shows the network which develops after fifty solo runs. The weights from this neuron to each of the neurons in the hidden layer are almost zero; the extra input thus has no ability to turn on or off the hidden neurons. However, there is a small weight on the direct connection to the output neuron. Even though this weight is small in comparison to the position and velocity direct connection weights (+1.6 vs. -8.5 and -7.1 respectively), it is large enough that some influence of the noise will be seen in the output. Further, since the bias on the noise will produce about -0.1 units of control, the rest of the network has clearly had to adjust and account for this term so as to maintain the correct amount of steady state control.

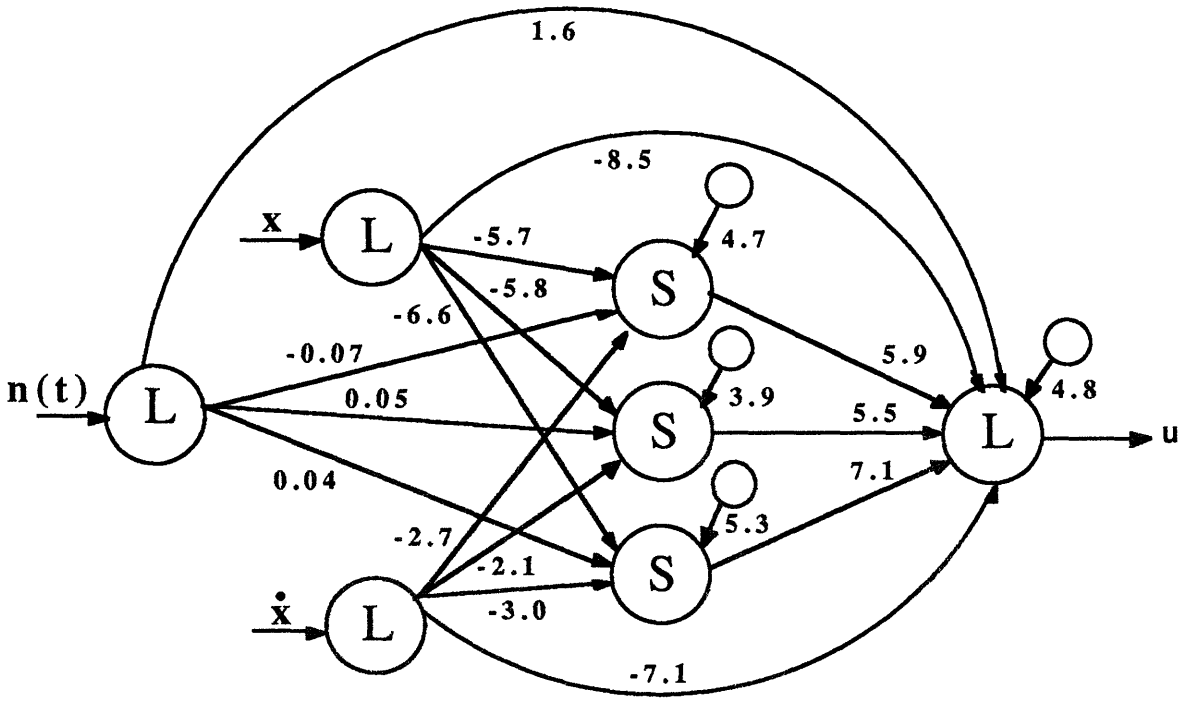

Figure 3.5.26: Network for extra sensor network after 50 runs

Figure 3.5.27 shows the response of the plant on the 50th solo run. The effect of the noise is indeed present, but barely noticeable. The response has a steady state mean of 1.002 and a standard deviation of 0.001. The switching laws which were developed are shown in Figure 3.5.28 and are not appreciably different from those developed during the canonical run. The simulation proceeds, as before, in the linear region of the hidden neurons.

Thus, except for the direct connection weight, the NMC algorithm has effectively cut the "useless" neuron off from the rest of the network. The direct connect weight has been kept small enough that the variance of the noisy sensor is virtually invisible in the output of the plant, and the bias has been offset by the actions of the other neurons. It would seem, based upon this experiment, than the NMC is capable of distinguishing between "relevant" and "irrelevant" inputs, and adjusting its weights accordingly. Interestingly, however, the control law that the NMC has developed now actually *depends* upon the presence of the extra neuron. If this neuron were to be removed from the network, or equivalently if its input were clamped to zero, the plant would stabilize around a slightly different equilibrium point, since the rest of the network is configured to offset the bias contributed by the extra neuron.

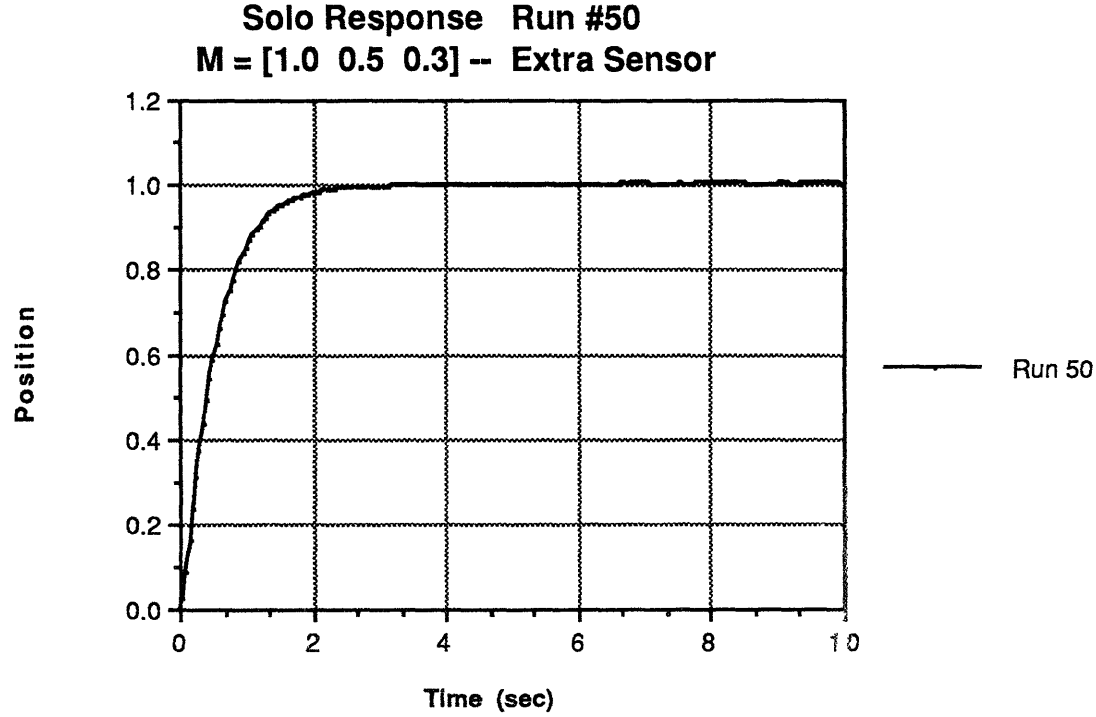

Figure 3.5.27: Solo response after fifty runs: extra sensor experiment

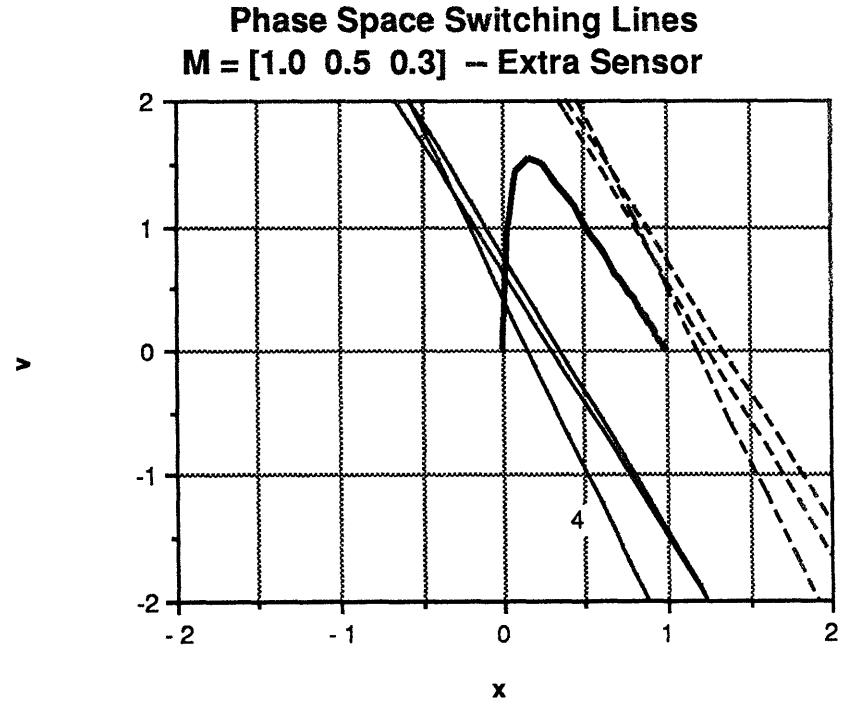

Figure 3.5.28: Switching lines implemented by network of Figure 3.5.27

Chapter 4: Further NMC Results

In this chapter several further results, obtained using simulations of the NMC algorithm on different SISO plants, are presented and analyzed. Again, unless otherwise explicitly noted, the state weighted form of the payoff function (equation 3.7) was used with the weighting vector $\mathbf{m}^T = [1.0\ \ 0.5\ \ 0.3]$, $n = 4$, and $u_{ult} = 16.0$. These analyses will necessarily be somewhat more abbreviated than those of the previous chapter; clearly different choices of the algorithmic parameters will result in different types of solutions (or lack thereof!) to the control problem. The results in this chapter serve only to demonstrate that the NMC algorithm can successfully function with a wide variety of plant dynamics.

Section 4.1 examines different linear plants, while Section 4.2 examines systems with either nonlinear (bang-bang) actuators, or else with nonlinear plant dynamics. Finally, Section 4.3 summarizes the observed cases where the NMC algorithm was found *not* to converge, and analyzes, in each case, why this might have been and how the algorithm could be modified to accommodate these cases.

4.1 Linear System Results

4.1.1 Output Regulation

This experiment sought to determine how the NMC algorithm would function when the variable to be regulated was not one of the states of the plant, but in fact a linear combination of plant states, in this case the velocity plus twice the position. Thus the plant transfer function:

$$G(s) = \frac{s+2}{s^2} = \frac{y(s)}{u(s)} \tag{4.1}$$

was used. The structure of the algorithm is not modified except that the measured variable, y, replaces the position state in all previous equations; this change is diagrammed in Figure 4.1.1. The desired state vector specifies the desired steady state *output*, as well as the desired final velocity, hence now $\mathbf{x}_d^T = [y\ \ \dot{x}]$, and the first entry of $\mathbf{m}$ weights deviations of the output from the desired equilibrium. The payoff function is similarly now a function of the output instead of the position state. Notice that, despite the fact that the

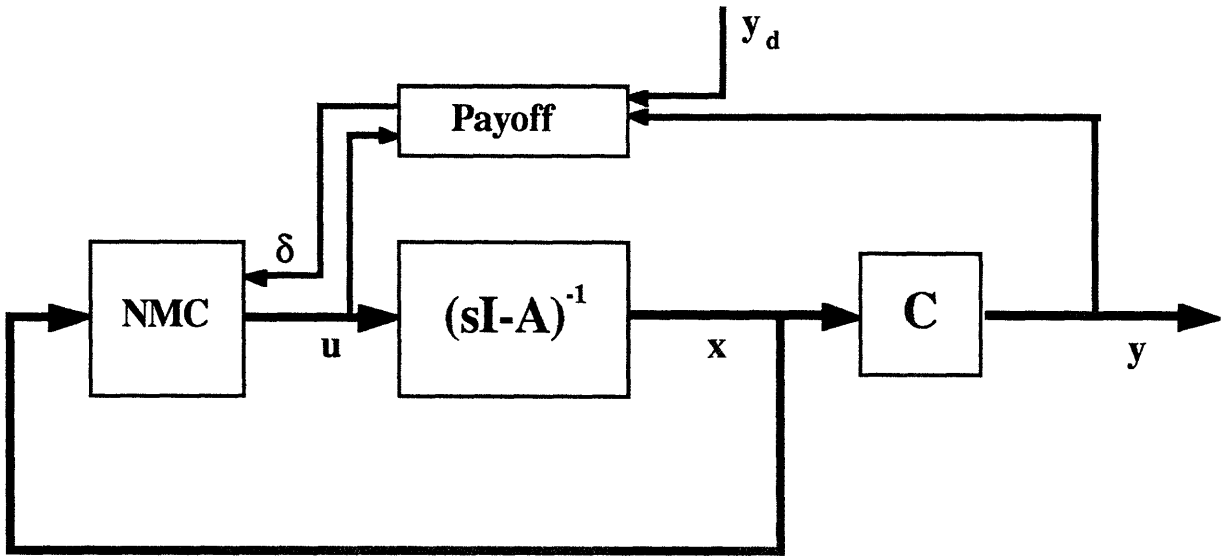

Figure 4.1.1: Structure of output regulation experiment

payoff function is now a function of y instead of x, the network is still shown the actual states of the plant; the NMC algorithm described in this thesis is thus a full state feedback algorithm.

The response after the fiftieth solo run is illustrated in Figure 4.1.2, and the network configuration and resulting switching lines are shown in Figures 4.1.3 and 4.1.4 respectively. Notice that the network has not only correctly determined that the correct final position state should be 0.5 (note that the position state does *not* explicitly appear in the payoff function for this experiment), it has also produced an overdamped response for the output, y(t). The control law, as is evident from Figure 4.1.4, is almost completely linear (recall that the control law is nonlinear only if the state trajectory crosses a switching line during the simulation), and is well approximated by $u(t) = 40.9(0.5 - x) - 32.1\dot{x}$, which results in closed loop poles at $s_1 = -30.8$ and $s_2 = -1.3$. Since the network has not succeeded in canceling the plant zero at $s = -2.0$, in order to maintain an overdamped response it has developed a control law which produces a closed loop exponential mode with a rather slow time constant. The combination of the closed loop zero and slow closed loop pole account for the rapid initial rise, then slower exponential settling of the observed response.

Clearly, different values of M_y or $M_{\dot{x}}$ will, as noted in Chapter 3, result in different control laws and different closed loop responses. In particular, the fact that velocity deviations are, in a sense, penalized twice in the payoff function--once through $y = \dot{x} + 2x$, and once through the direct velocity weighting--is probably the cause of the (perhaps excessively) overdamped solution obtained by the network for this experiment.

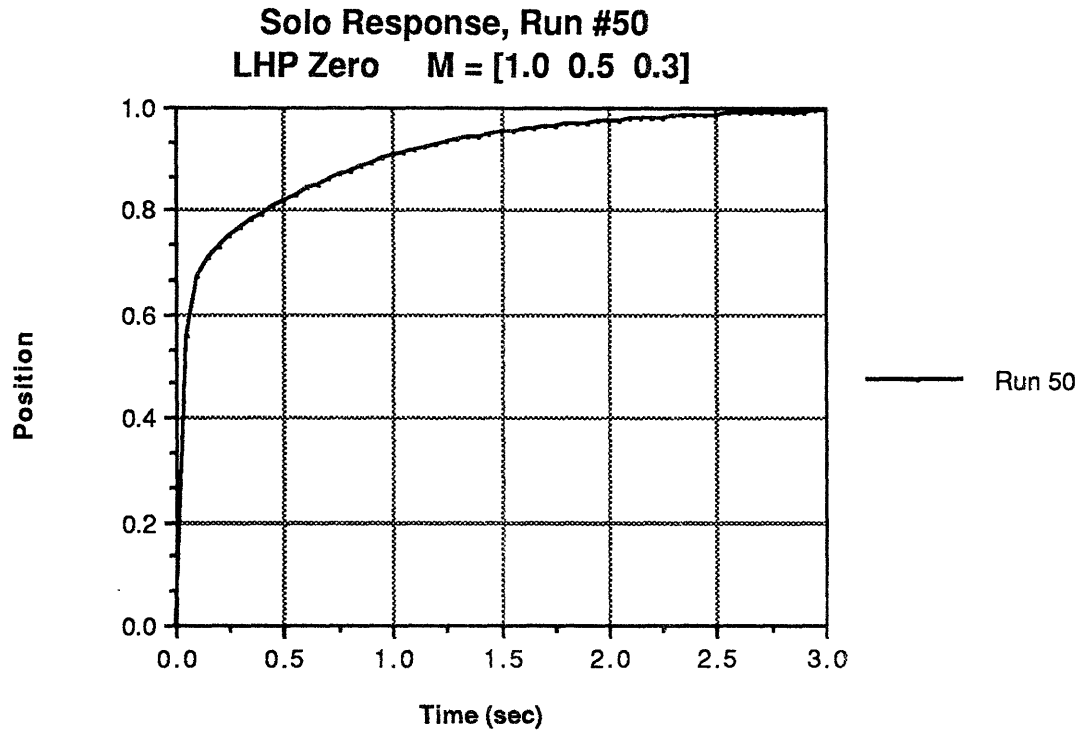

Figure 4.1.2: Solo response, run #50 for the output regulation experiment

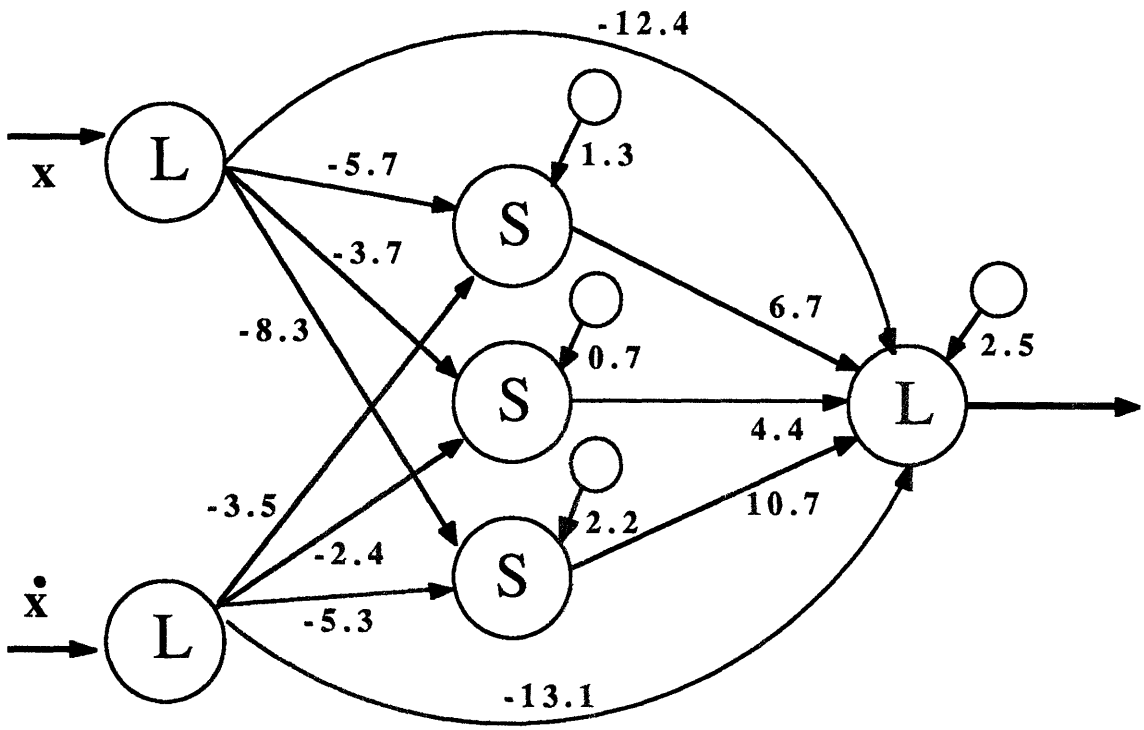

Figure 4.1.3: Network configuration after 50 iterations for output regulation experiment

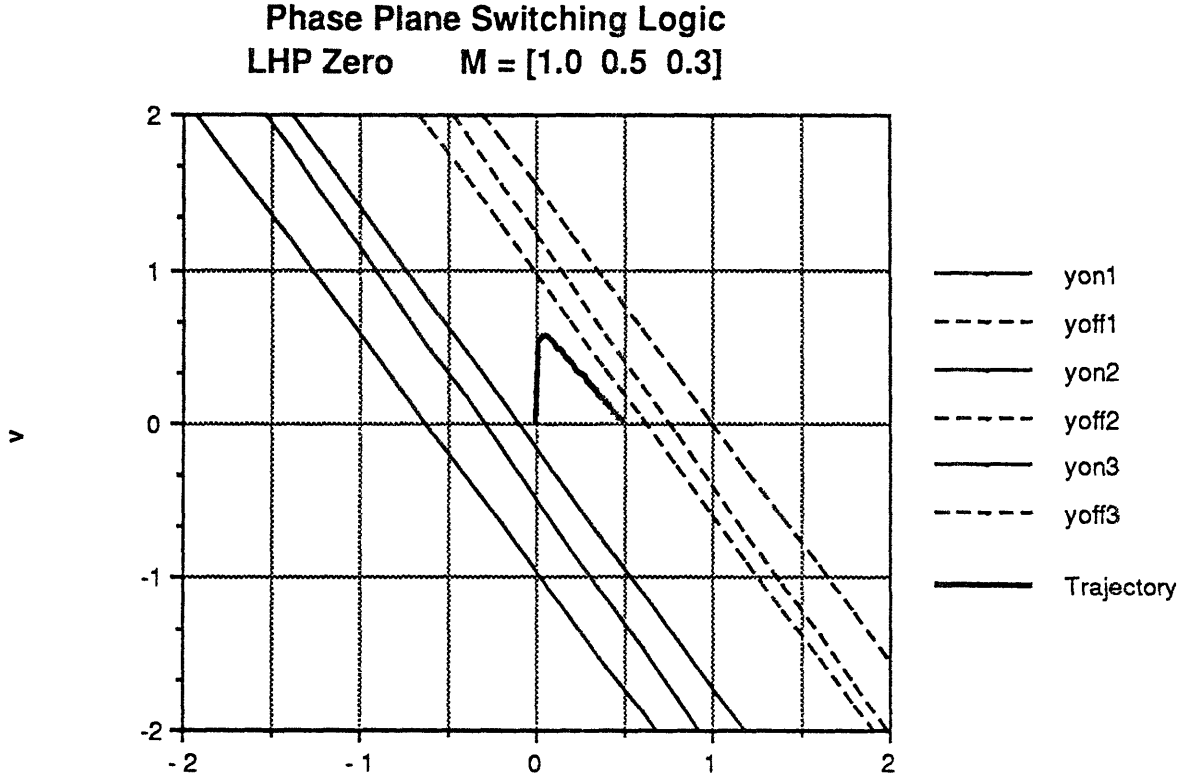

Figure 4.1.4: Switching logic implemented by network of Figure 4.1.3

4.1.2 Open Loop Unstable Plants

For this experiment, the unstable plant:

$$G(s) = \frac{3}{(s + 3)(s - 3)} \tag{4.2}$$

was used in the simulations. In fact, the network has already solved the control problem posed by this plant in Section 3.5.1, but in that experiment the network had already developed a negative feedback control scheme at the time it was exposed to the unstable plant. Here the objective is to evaluate whether the algorithm can develop the correct controller starting with a "blank" network.

Figure 4.1.5 shows the closed loop response which is obtained after fifty iterations. Clearly a stabilizing controller has been constructed and, as is obvious from Figure 4.1.6 - 4.1.8, the control law is essentially linear over the states experienced in the simulation. Even more interesting is the fact that the network has determined the amount of control required to maintain the plant at the desired equilibrium position, in this case -3.0 units. This is significant because there is nothing in the payoff function which explicitly tells the network the amount of steady state control required; the network has ascertained this information solely on the basis of its interactions with the plant dynamics.

The fact that a finite amount of steady state control is required means that the algorithm will not be able to both drive the payoff function exactly to zero and maintain the desired position equilibrium. This suggests that problems may arise as the amount of control required to maintain the equilibrium position approaches u_{ult}; for such cases the contribution of the control term to the magnitude of the payoff function will become quite significant. As the next two sections will demonstrate, the algorithm can be quite ingenious in dealing with this problem. However, for open loop unstable plants the problem is more severe, and can lead to instabilities in the training process. Section 4.3 presents a more complete discussion of this, and other, instabilities.

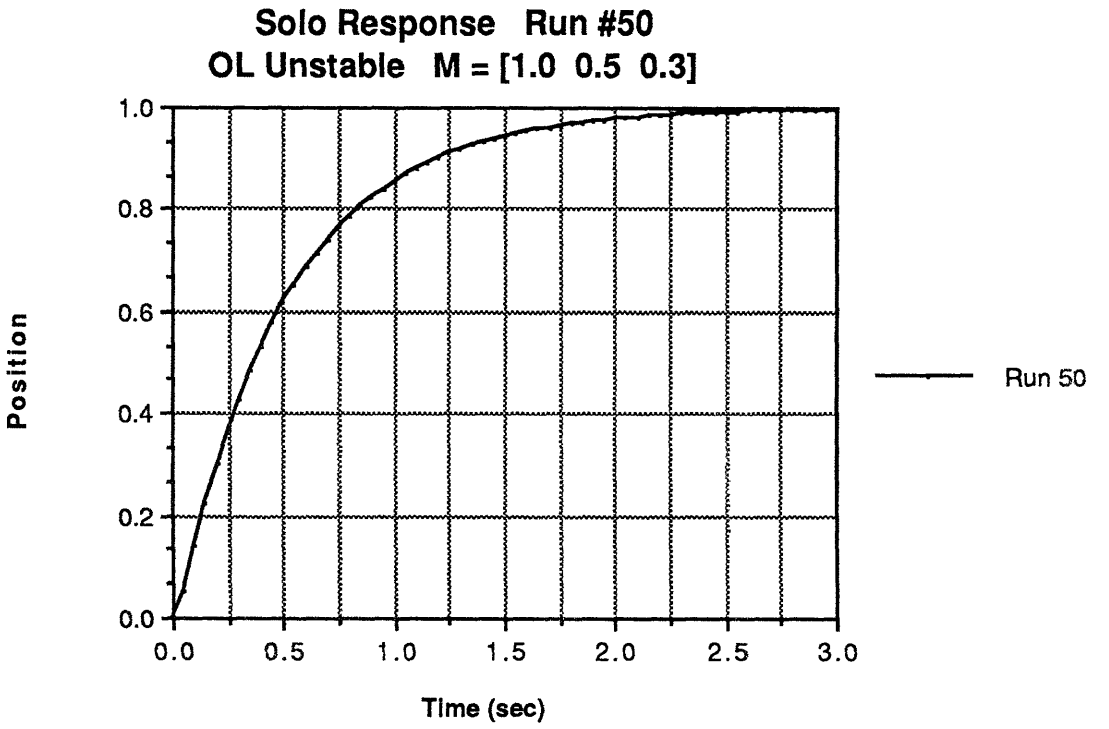

Figure 4.1.5: Solo response after 50 iterations for open loop unstable plant

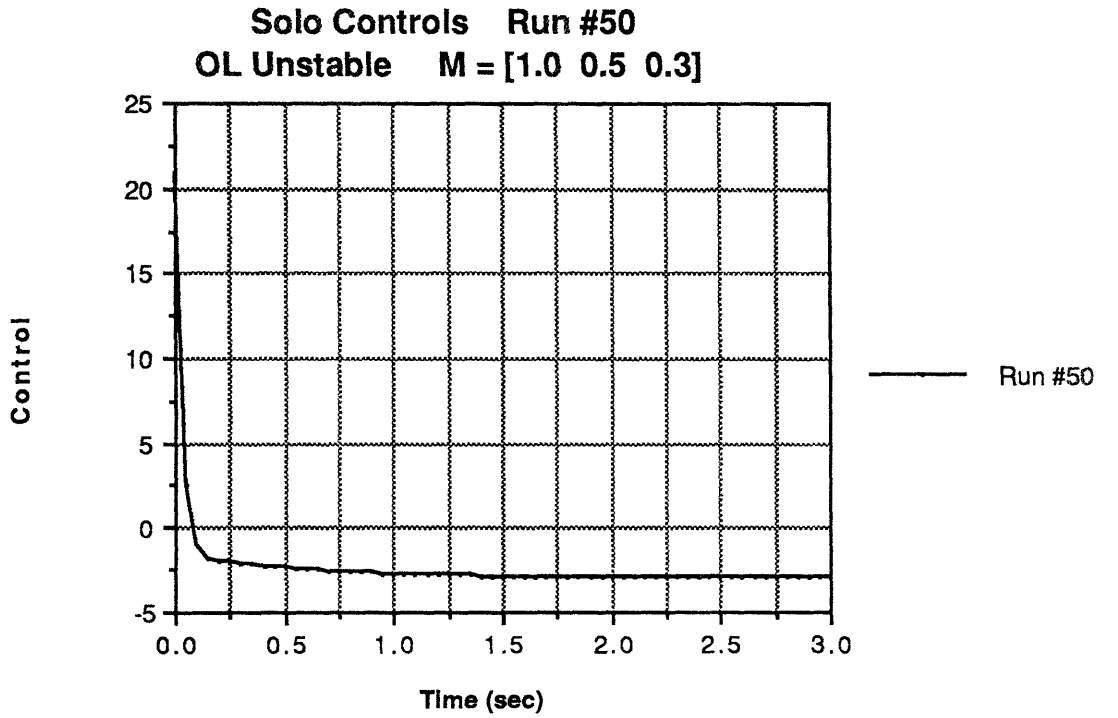

Figure 4.1.6: Controls commanded by the network, 50th iteration: open loop unstable plant

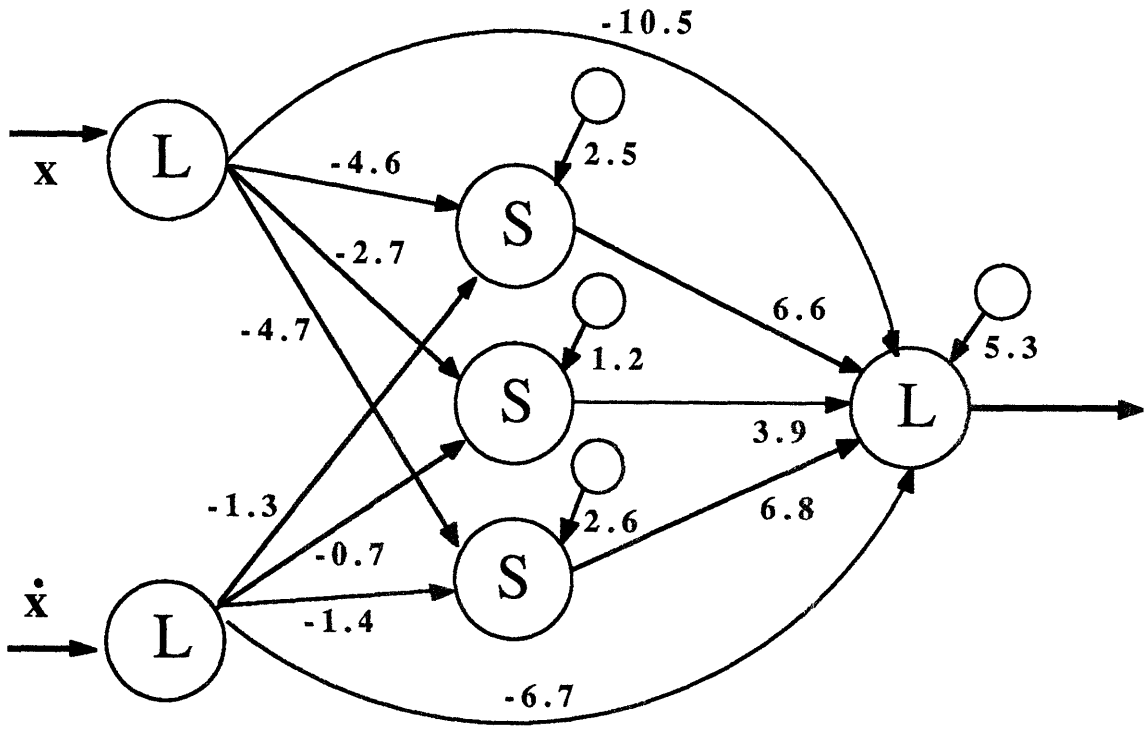

Figure 4.1.7: Network after 50 iterations for open loop unstable plant

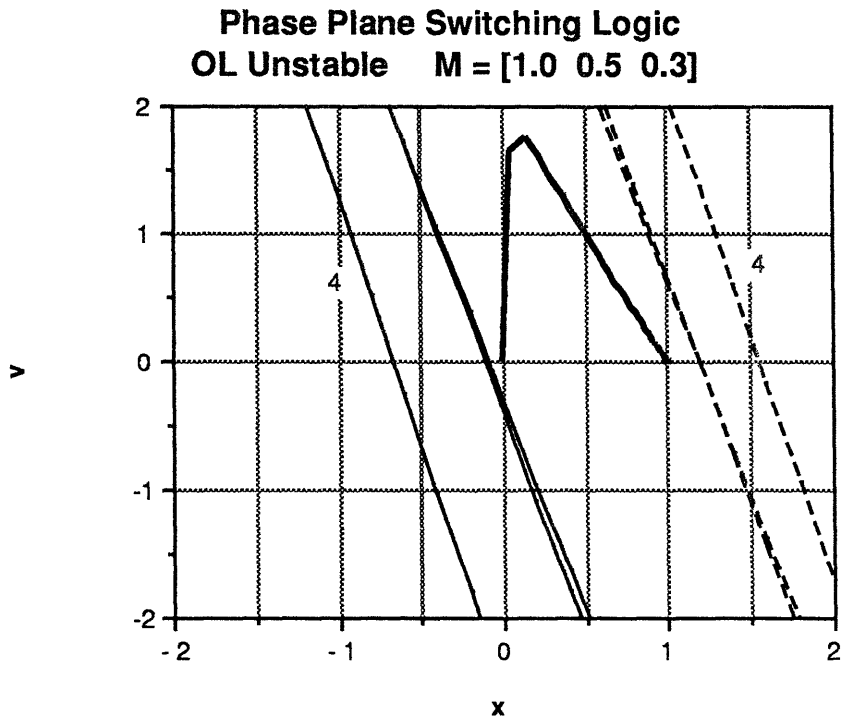

Figure 4.1.8: Switching logic implemented by network of Figure 4.1.7

4.1.3 A Simple Harmonic Oscillator

For this next experiment, the open loop plant was chosen to be the undamped oscillator:

$$G(s) = \frac{1}{(s^2 + 12)} \tag{4.3}$$

This plant was deliberately chosen to explore what, if any, tradeoff the NMC algorithm could make when the required steady state control makes a nontrivial contribution to the payoff function. Figures 4.1.9 through 4.1.12 summarize the closed loop response, control signals, network configuration, and switching logic which developed after fifty iterations. Notice that the network does *not* successfully stabilize the plant about the $x_d^T = [1.0\ \ 0.0]$ equilibrium state. In fact the observed equilibrium is $x^T = [0.93\ \ 0.0]$. Despite this, the control law is again almost completely linear (except just as the plant begins to move from the rest position), producing the overdamped response observed in Figure 4.1.9

But why has the algorithm stabilized about this equilibrium point in particular? From examination of equation (3.7) and the above discussions, it has been determined that the algorithm operates by attempting to drive $\delta(t)$ to zero at each instant in time. However, the form of the state weighted payoff function is such that, if the desired equilibrium were maintained for this system, a value of:

$$\delta_{ss} = -M_u(u_{ss}/u_{ult})^n = 0.3(12.0/16.0)^4 = -0.092 \tag{4.4}$$

would obtain. At the observed equilibrium, however, where the observed $u_{ss} = 11.15$, one obtains:

$$\begin{aligned} \delta_{ss} &= M_x(1.0 - x_{ss}) - M_u(u_{ss}/u_{ult})^n \\ &= 1.0(1.0 - 0.93) - 0.3(11.15/16.0)^4 \\ &= -0.0007 \cong 0 \end{aligned} \tag{4.5}$$

The algorithm has thus performed a tradeoff between steady state accuracy and steady state control authority. Since the contribution of each term to the magnitude of the payoff function is indistinguishable to the network, there is no special significance is given to

maintaining the desired steady state equilibrium; the final state achieved by the controller implemented by the network is treated as a variable in the minimization of $\delta(t)$.

In fact, in this example at least, it is easy to show that this tradeoff is *optimal* with respect to minimization of δ_{ss}. The steady state equilibrium which will result can thus be determined as the solution of:

$$x_{ss} = \min_{x} \{|\delta_{ss}(t)|\} \tag{4.6}$$

$$= \min_{x} \left\{ \left| M_x(x_d - x) + \mathrm{sgn}(u) M_u \left| \frac{u_{ss}}{u_{ult}} \right|^n \right| \right\} \tag{4.7}$$

or, assuming the canonical weighting parameters and a second order system:

$$= \min_{x} \left\{ \left| (x_d - x_{ss}) - \frac{\dot{x}_{ss}}{2} - 0.3 \left| \frac{\lambda x_{ss}}{16} \right|^4 \right| \right\} \tag{4.8}$$

where λ is the inverse of the DC gain of the plant (note that if the DC gain is infinite, i.e. the plant has one or more integrators, the control term will drop out of (4.8) just as desired). Given the dynamics of the system, δ can assume a minimum only if $\dot{x}_{ss} = 0.0$, although of course the algorithm does not know this *a priori*; the fact that this feature of the relations between the plant states has been determined is itself interesting and will be explored in more detail in the next section. A plot of (4.8) for this particular plant, with $\lambda = 12.0$ and $\dot{x}_{ss} = 0.0$, versus x is shown in Figure 4.1.13. The minimum is at $x = 0.93$, exactly the observed steady state value.

These results imply that, based upon its interaction with the unknown dynamic system, not only can the network correctly determine the required amount of steady state control, but as this control begins to approach u_{ult} the network can also make an optimal tradeoff between tracking error and required control authority, with the condition of optimality being minimization of δ_{ss}.

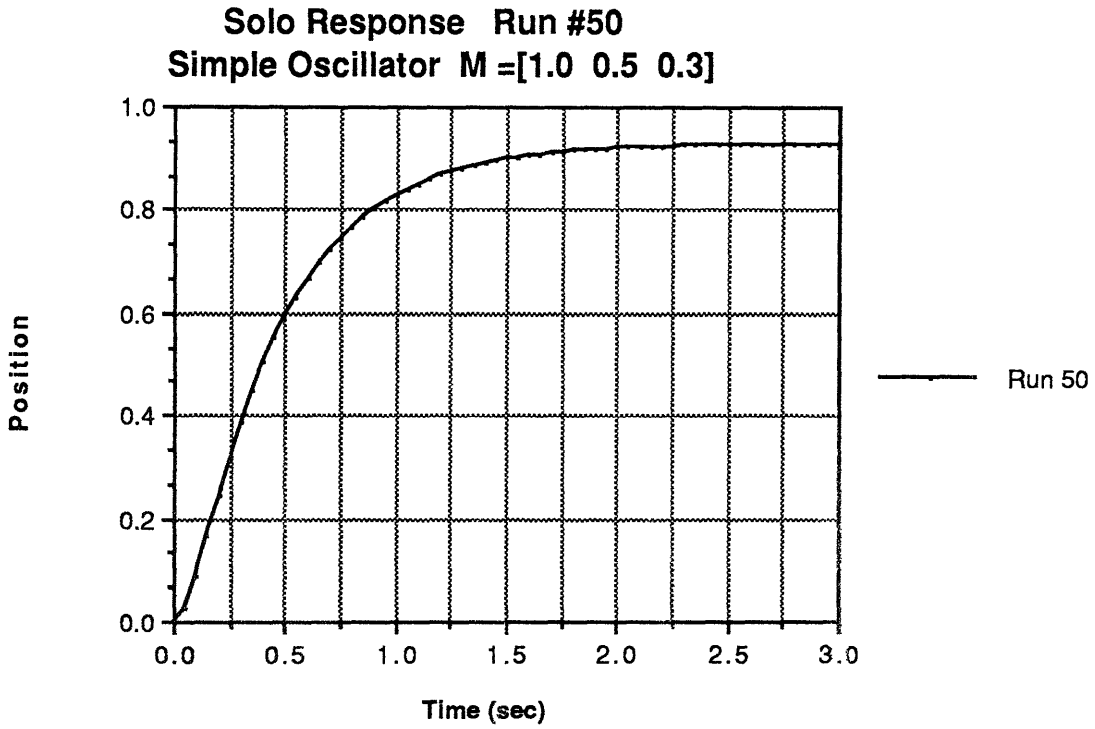

Figure 4.1.9: Solo response after 50 iterations: simple oscillator experiment

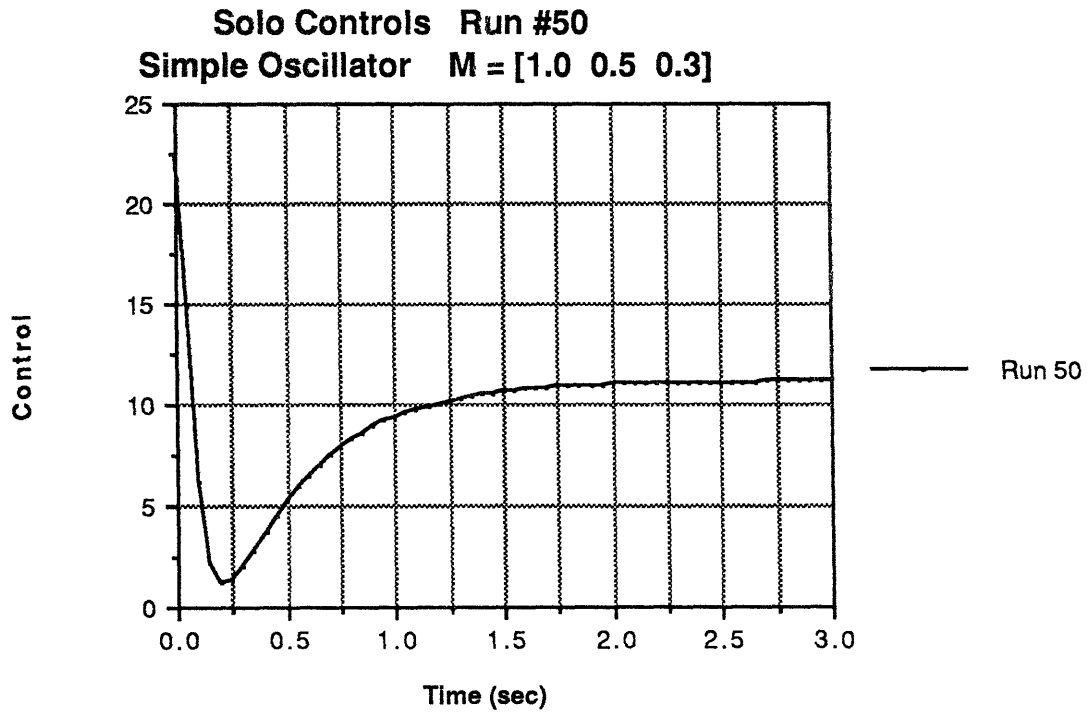

Figure 4.1.10: Controls commanded by the network for simple oscillator experiment

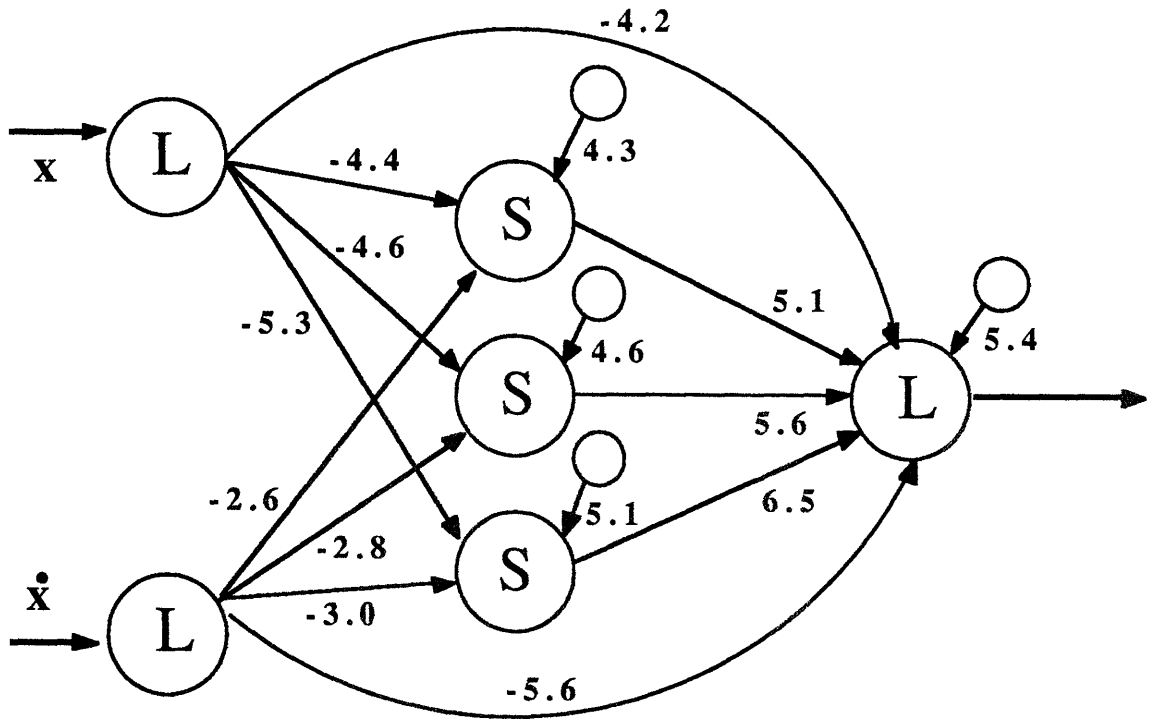

Figure 4.1.11: Network after 50 iterations: simple oscillator experiment

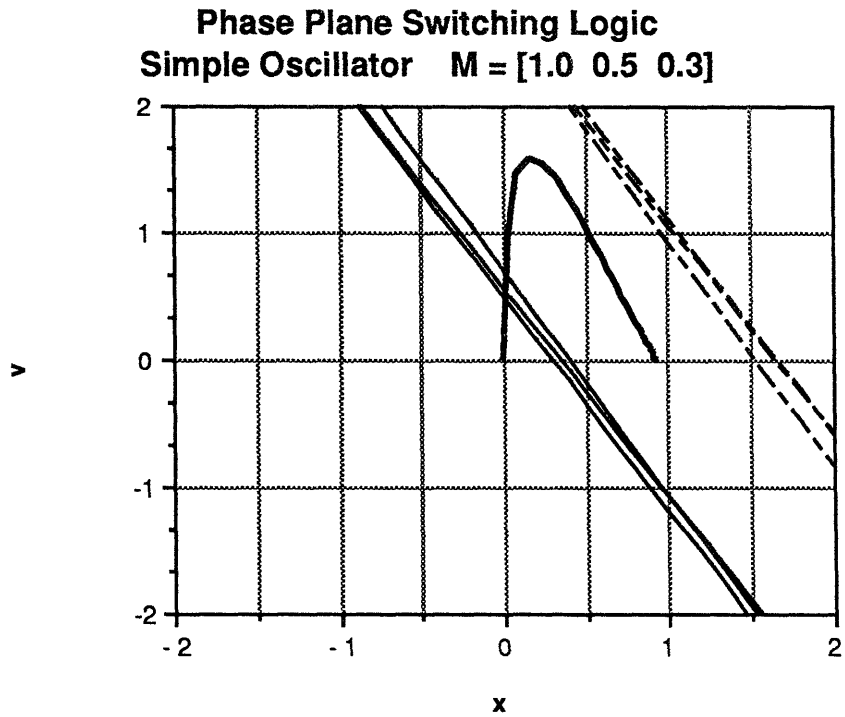

Figure 4.1.12: Switching logic implemented by network of Figure 4.1.11

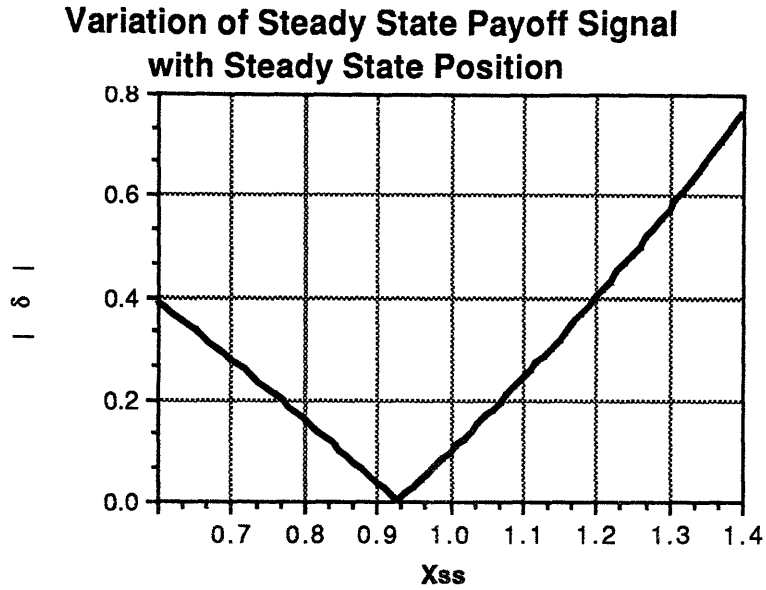

Figure 4.1.13: Plot of equation (4.8) for the plant of equation (4.3). Notice the minimum is at the observed steady state position shown in Figure 4.1.9

4.1.4 Velocity Regulation

To test the generality of the algorithm with respect to different desired plant states, an experiment was performed with the double integrator plant for which $\mathbf{x}_d^T = [X \;\; 1.0]$ where X indicates this value of the desired state is meaningless. The algorithm is thus being requested to construct a velocity regulator instead of the position regulator examined in most of the above experiments. To this end, the weighting matrix $\mathbf{m}^T = [0.0 \;\; 1.0 \;\; 0.3]$ was used along with the state weighted payoff function (3.7). As Figures 4.1.14 through 4.1.16 demonstrate, the algorithm has had no trouble developing the desired controller. However, two items deserve special note. First, in order for a velocity regulator to work for arbitrary position deviations from rest, there should be *no* weight on the direct linear position synapse in the network and zero weights on the position state neuron's connections to the hidden neurons, i.e. W_1 through W_4 should be identically zero. As Figure 4.1.15 shows, this is almost, but not quite, the case; each of these weights is very small, but not exactly zero. The influence position deviations exert on the control will hence be quite small only for values of x less than approximately 10.0; values larger than this and the position deviations will begin to interfere with the velocity regulator. The fact that these weights are not identically zero is another manifestation of the limited training set the network experiences while it learns. Since each training phase lasted only from ten to twenty simulated seconds, during which the position deviations never

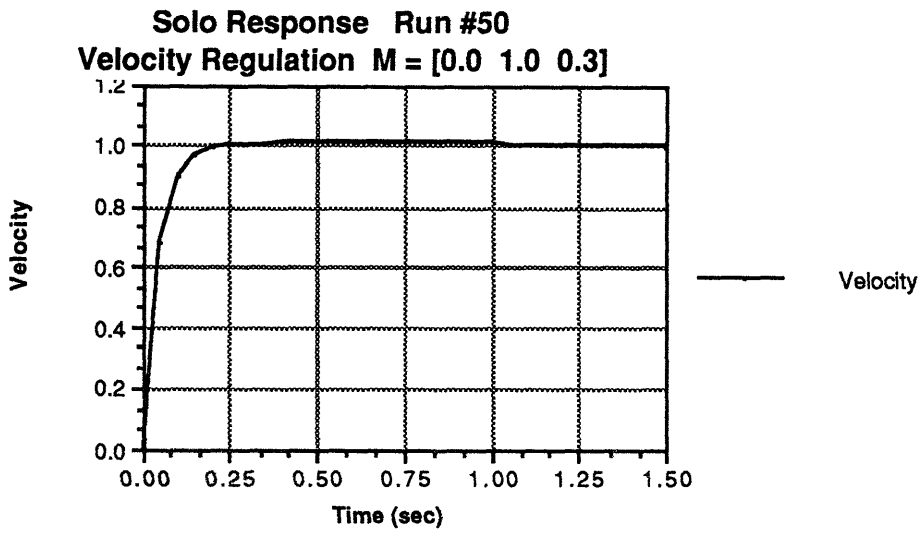

Figure 4.1.14: Solo response after 50 iterations: velocity regulation experiment

grew larger than about 20.0, the weights from the position neuron were small enough that they did not cause problems for the values of state experienced while training.

The second observation is the shape of the switching lines. These are quite different than those observed in the previous simulations, not only in orientation, but also in the huge width of the linear regions. The corresponding off switching boundaries for the on switching lines shown in Figure 4.1.16 parallel the shown lines, but intercept the velocity axis between -8.0 and -11.0, far off the bottom of the region shown in the diagram. Each of the lines is (almost) parallel to the position axis, reflecting the above observation that the position states have little or no impact on the hidden neurons.

An interesting question which arises in conjunction with the velocity regulator is what would happen if the algorithm were given conflicting instructions, for example, if it were told to maintain a specified position equilibrium *and* a nonzero velocity equilibrium. Such an experiment was conducted with $\mathbf{m}^T = [1.0\ 0.5\ 0.3]$ and $\mathbf{x_d}^T = [0.0\ 1.0]$. Clearly, with this combination of desired state and state weightings, the network will have to sacrifice either the position or the velocity regulation tasks. Further, this "decision" will have to be made in conjunction with the system dynamics, since the network does not know, *a priori*, that nonzero velocity deviations will produce an infinitely increasing position deviation and hence, given the weighting vector, an infinitely increasing $\delta(t)$. As might be expected from the results of the last section, the algorithm arrives at a final state which ensures that δ_{ss} is minimized, in this case $\mathbf{x_{ss}}^T = [0.5\ 0.0]$, which, it can be easily verified, results in $\delta_{ss} = 0$. Thus, even though the "problem" given to the algorithm was ill posed, the algorithm was capable of designing a network which optimally satisfied the constraints.

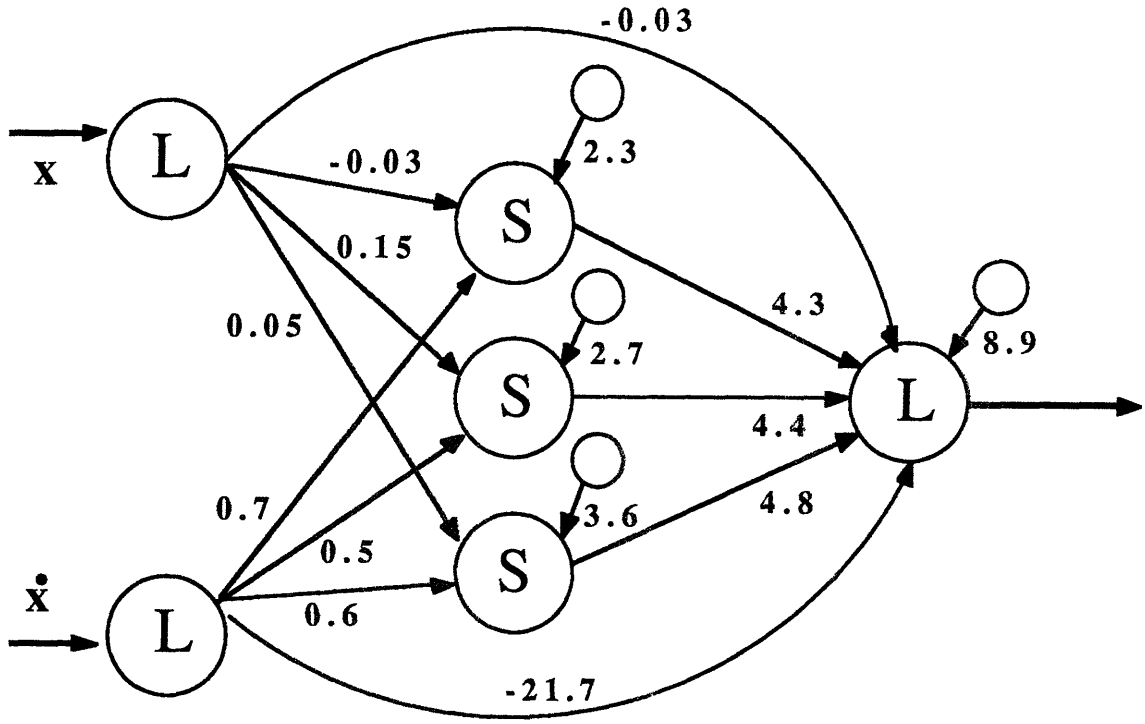

Figure 4.1.15: Network after 50 iterations: velocity regulation experiment

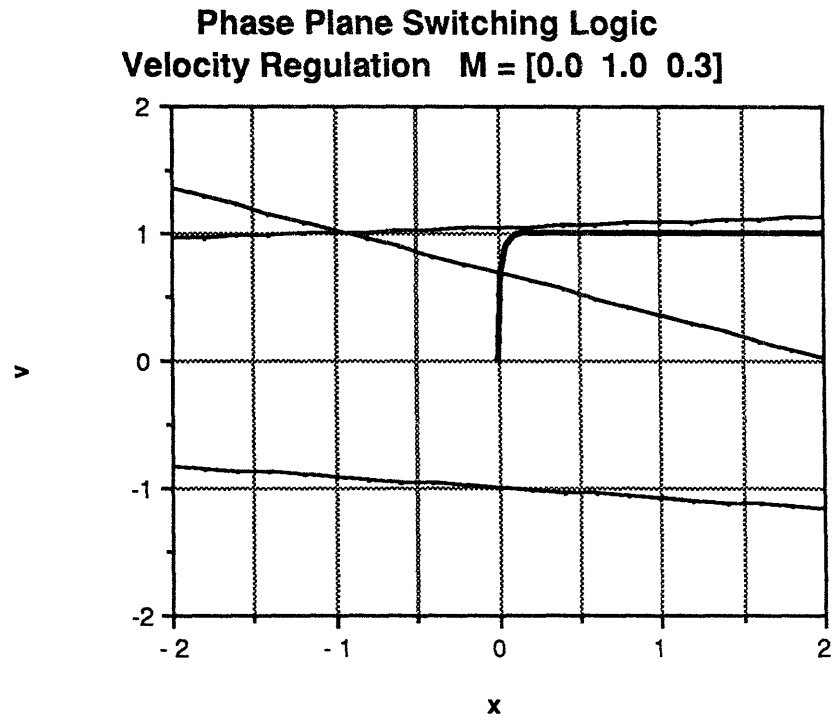

Figure 4.1.16: Switching logic implemented by the network of Figure 4.1.15. See text for a detailed explanation

4.1.5 Random Initial Plant Conditions

A significant question which has been raised by the results of the preceding sections is to what extent the control laws developed by the network depend upon the states experienced while training. While it is impossible (and probably undesirable) to have the network experience the infinitude of possible plant states, it is possible to modify the training process so the system at least encounters a wider variety of initial states. To this end, the canonical run using the double integrator plant of Section 3.3.3 was repeated, but starting from different initial conditions during each training phase. The initial conditions for each phase were randomly determined using the pseudorandom number generator discussed in Section 3.5.3; the initial positions were thus uniformly distributed on the interval [0.0 2.0], and the initial velocities were uniformly distributed on the interval [_1.0 1.0]. To allow for these widely different initial conditions, the algorithm was allowed to run for 100 iterations.

Figure 4.1.17 displays the solo response after the 100th iteration. The initial conditions for this run were $\mathbf{x}_0^T = [1.52\ \ 0.90]$. Note that, after the initial velocity has been overcome, the response settles exponentially to rest with approximately the same time constant as the canonical run. As a more explicit comparison, Figure 4.1.18 shows the response obtained with the same network but initial conditions of $\mathbf{x}_0^T = [0.0\ \ 0.0]$ plotted against the canonical response obtained in Section 3.3.3; the two responses are almost identical, as expected. The network which implements the controller and the resulting switching lines are shown in Figures 4.1.19 and 4.1.20 respectively. Notice that the linear part of the control law is similar to the canonical run except for the position feedback gain which is substantially larger (-14.6 vs. -8.6 for the canonical run). Comparing the switching lines with those of Figure 3.3.18 reveals a few significant differences. The switching lines of neurons three and five have been pushed further to the right; the switching line for neuron four has been rotated 90 degrees clockwise making it perpendicular to those of neurons three and five, and its linear region has been expanded by almost a factor of two.

At first glance some of these switching lines may appear counterintuitive. For example, why should neuron four switch on, introducing +3.5 units of control, for large positive velocity deviations, and why should neurons three and five switch off for very large positive position and velocity deviations; it seems the correct responses in these cases would be exactly the opposite. The answer to this lies in the tradeoff in $\delta(t)$ being accomplished by the algorithm; the network has been arranged to *both* maintain the desired equilibrium *and* keep the control within the established guidelines. The linear half of the

control law is sufficient to stabilize the plant about the desired equilibrium, but for large deviations the linear terms may cause the control to approach or exceed u_{ult}. To offset this, the network can use its switching lines to actually oppose the linear terms, in regions far from equilibrium, and hence reduce the amount of control used in those regions.

The results of this experiment are encouraging in the sense that many of the features of the original switching logic have been retained despite the radically different training environment. The equilibrium point still lies near the center of the linear regions of all the hidden neurons, for example, and the orientation and position of two of the three switching lines have remained essentially unchanged. Even still, the fact that the switching lines have changed at all as a result of the increased training set raises an issue which is common in adaptive control or systems identification theory: the concept of *persistency of excitation* or *sufficient richness*. Put simply, in order for any adaptive architecture to function properly, the plant must be so excited as to reveal all of the salient characteristics of its dynamics. If the plant moves through its state space in a "boring" manner, i.e. one which is not truly characteristic of its inherent dynamics, an adaptive algorithm may develop a control strategy which would result in very poor performance, perhaps even instability, when more "stimulating" trajectories are commanded.

Here this sufficient richness condition manifests in the recurring question as to whether the range of states, and hence the range of plant characteristics, experienced is adequate to allow the network to develop a control law which would be valid even for states not encountered during the training periods. In most of the cases examined so far, the answer to this question is affirmative. The important features of the control laws developed in those experiments were the negative linear feedback terms; these terms will dominate in states far from the equilibrium, since the linear input neurons have unlimited dynamic range, whereas the hidden sigmoidal neurons are limited to the interval [0, 1]. In two cases, however, in the *exp(-t)* model trajectory, and in the state weighted velocity regulator experiments, control laws were constructed by the network which were valid only for the range of states experienced in the simulation. While randomizing the plant initial conditions used during training would clearly help this situation, and without adverse effects on the resulting closed loop response as this section has shown, it is always possible that the plant, or the control law, would exhibit unexpected behavior in regions of state space not visited during training. The key issue in determining this will be the extent to which the training set is sufficiently rich for both the plant and the network, i.e. if the training set is truly representative of all possible situations the controller must handle.

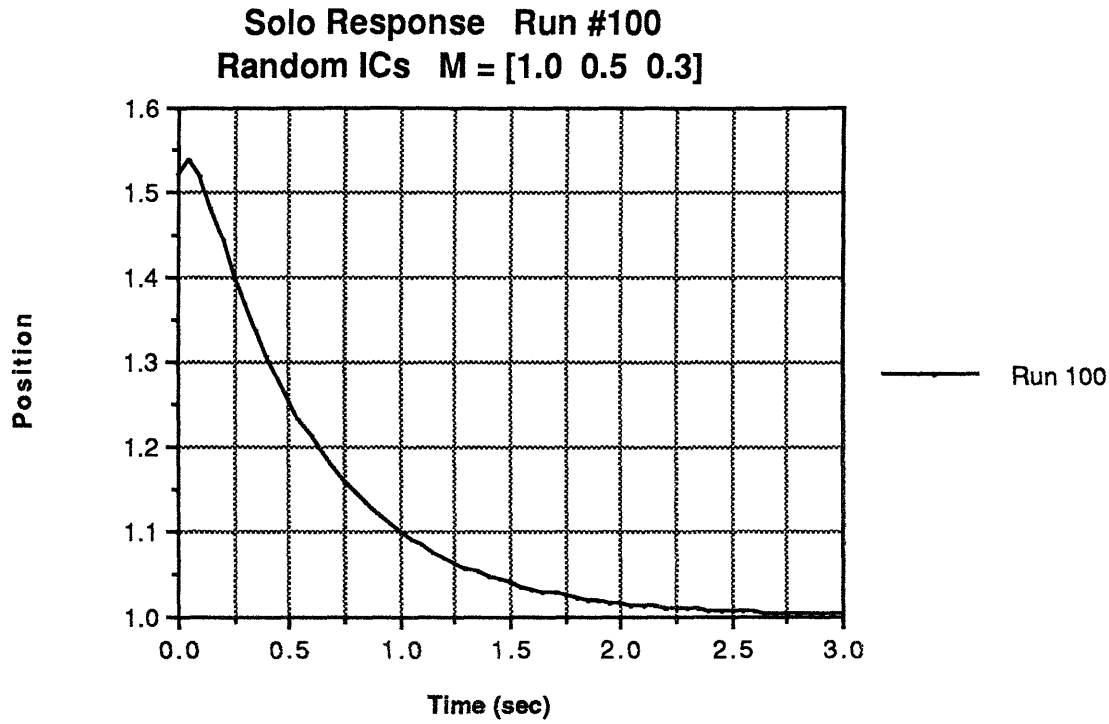

Figure 4.1.17: Solo response after 100 iterations; random initial conditions experiment

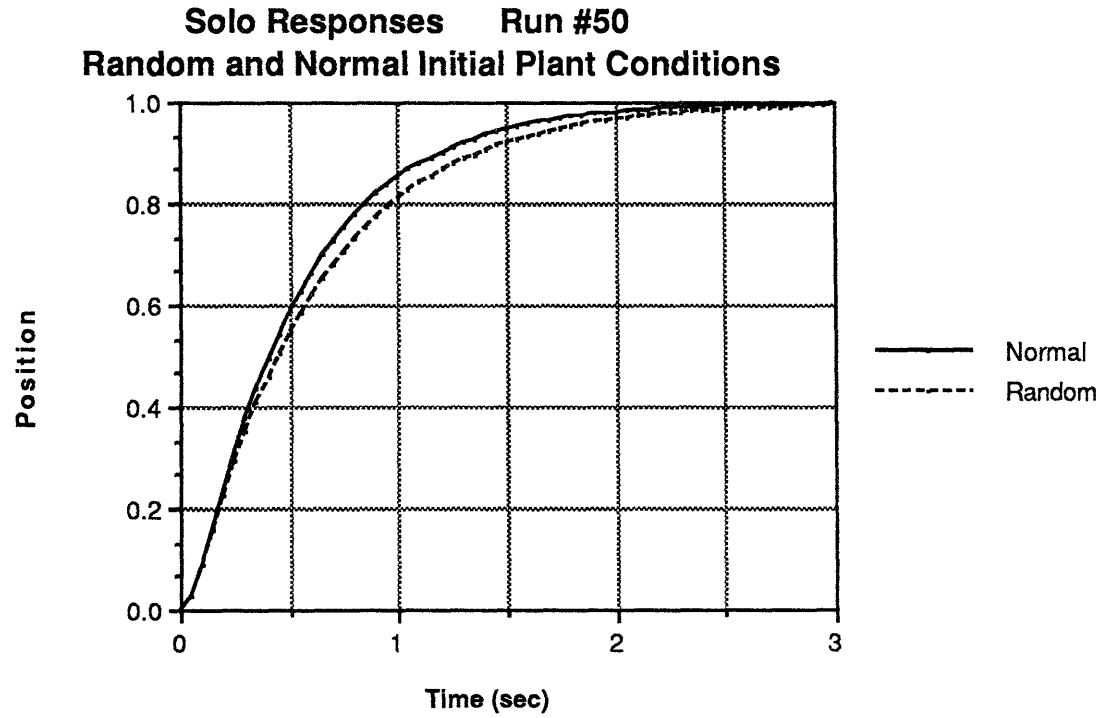

Figure 4.1.18: Comparison of responses from zero initial conditions obtained using canonical network configuration (q.v. Section 3.3.3) and a network trained with random initial plant conditions

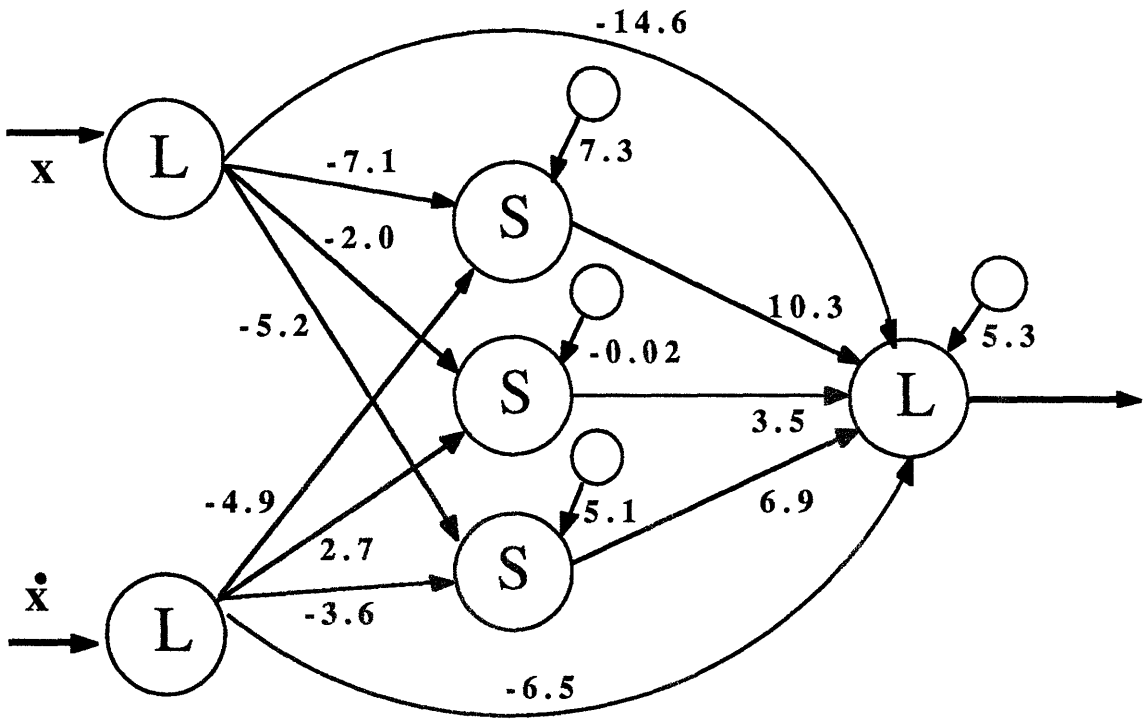

Figure 4.1.19: Network after 100 iterations: random initial conditions experiment

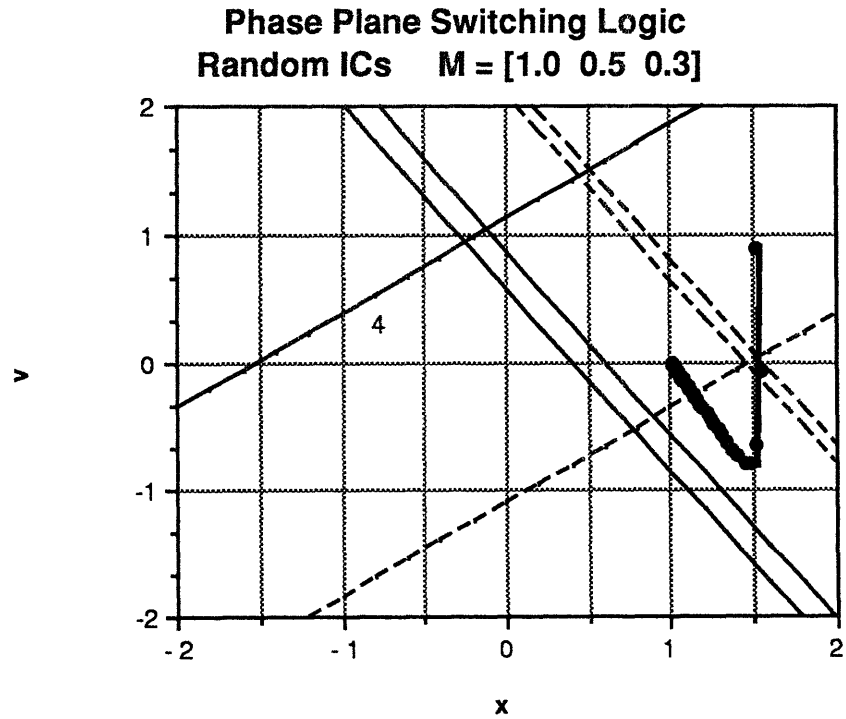

Figure 4.1.20: Switching logic implemented by the network of Figure 4.1.19.

4.1.6 Triple Integrator

The last experiment with linear systems was performed to evaluate the behavior of the algorithm with a higher dimensional plant, in this case a triple integrator. In keeping with the above acknowledgement that the NMC algorithm is essentially a full state feedback methodology, the network is shown all plant states; the third order NMC network shown in Figure 2.6.2 was thus used for this experiment with the desired equilibrium $\mathbf{x}_d{}^T = [1.0\ \ 0.0\ \ 0.0]$. The weighting vector was chosen to be $\mathbf{m}^T = [1.2\ \ 0.8\ \ 0.4]$; this was somewhat arbitrary, but was guided by the parameter tradeoff conducted in Chapter 3.

The solo responses shown after the fiftieth iteration are shown in Figure 4.1.21. The response is somewhat underdamped, overshooting the desired equilibrium position by about 9% before settling, even though the ratio of velocity to position state deviation weighting is greater in this case than in the double integrator problem. It would appear that in general the shape of the closed loop response will vary not only as a function of the weighting matrix used, but also as a function of the plant dimension; generalizations established for second order plants may not be transferable to third order dynamics. Nonetheless, the network is still implementing an effective, stabilizing position regulator.

Figure 4.1.22 shows the network which implements the control law. It is not possible to adequately display the switching logic (in this experiment they would be planes cutting through the three dimensional state space) for this system. However, analysis of the data reveals that the control signal shown in Figure 4.1.23 is well approximated by the linear state feedback $u = -[27.3\ \ 21.8\ \ 10.3]\mathbf{x}$, which yields closed loop poles at $s_{1,2} = -1.15 \pm 1.45j$ and $s_3 = -8.0$. This pole structure produces a step response which agrees exactly with that shown in Figure 4.1.21. Thus, even for this third order plant, the NMC algorithm has produced a network which implements a feedback law which is essentially linear, at least for the range of states experienced in the simulations.

These further results with linear plants confirm the observations of the previous chapter. The NMC algorithm constructs networks which implement control laws capable of stabilizing the (unknown) plant about the desired equilibrium. Further, quite often these control laws are completely linear over the range of states experienced in the simulation. As before, the observed control laws arise from the concerted actions of all the neurons in the network, especially in the instances where the control law is linear and the effective feedback gains are significantly greater than those which would be expected by looking at the direct linear terms.

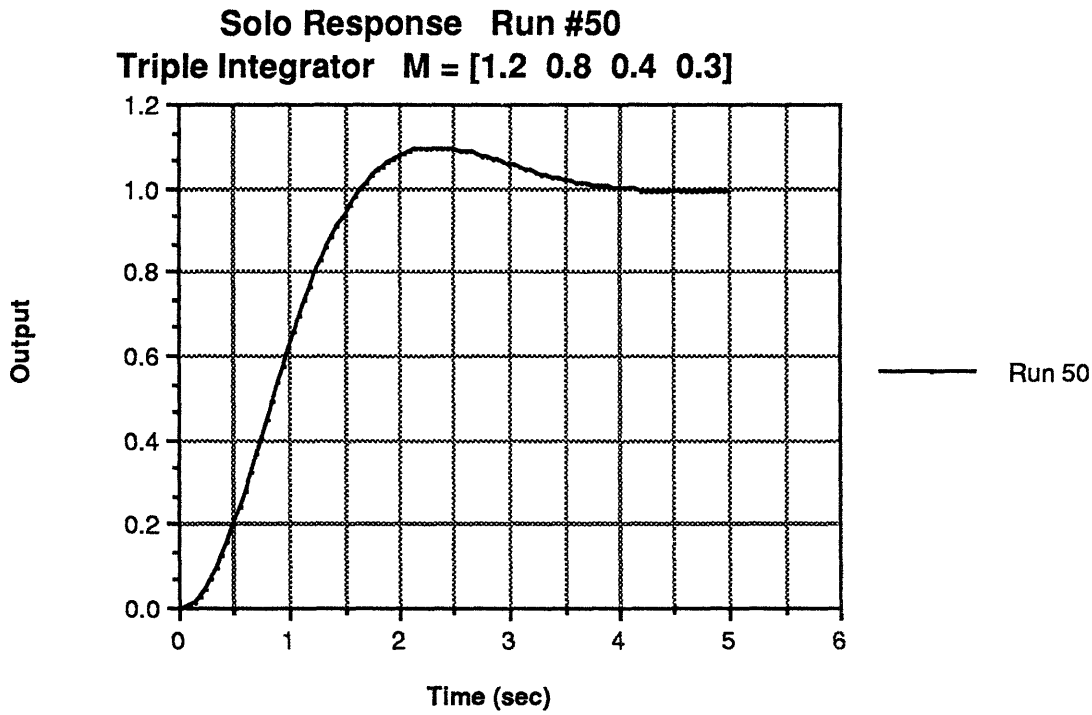

Figure 4.1.21: Solo response after 50 iterations: triple integrator experiment.

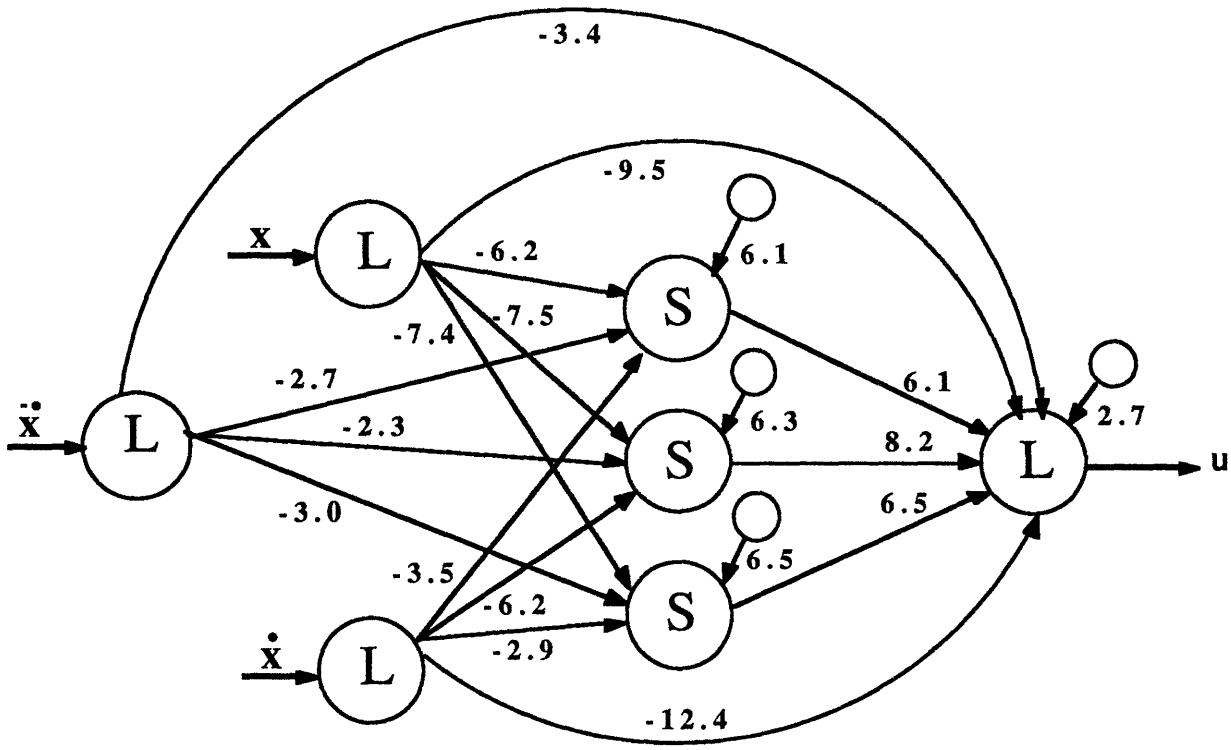

Figure 4.1.22: Network after 50 iterations: triple integrator experiment.

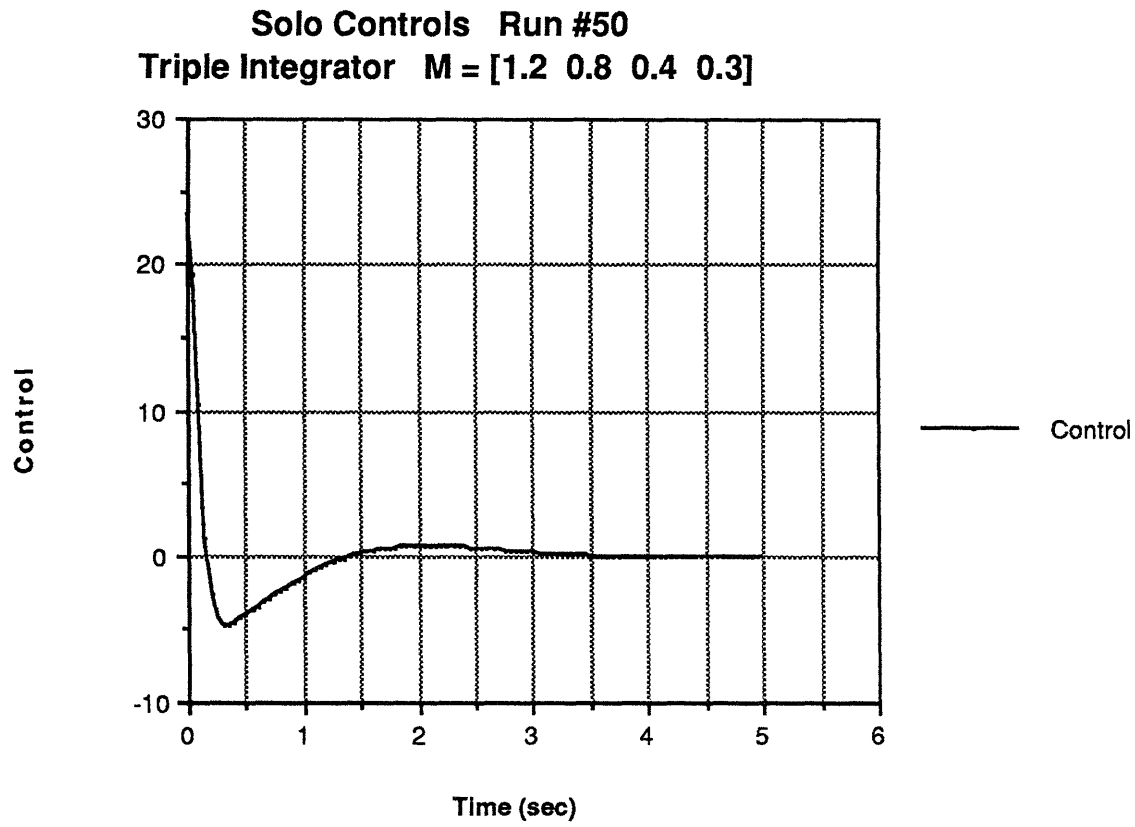

Figure 4.1.23: Control signals commanded by the network: triple integrator experiment.

4.2 Nonlinear System Results

Having shown that the NMC algorithm produces sensible controllers for second (and one third) order linear systems, it is necessary to evaluate the performance of the algorithm for nonlinear plants and actuators. The eventual application of this learning control algorithm in the SSL will be in the attitude and position control of the laboratory's underwater teleoperated devices. Such devices experience at least three kinds of nonlinearities in their dynamics: viscous water drag, actuators which saturate and which can provide forces only in quantized levels, and finally the "pendulum" effects resulting from non-collocation of the centers of gravity and buoyancy in the vehicle. The next three sections examine the ability of the algorithm to construct controllers in the face of each of these nonlinearities, concluding with a full nonlinear simulation of the pitch attitude dynamics of one of these teleoperated vehicles.

4.2.1 Viscous Drag

Viscous drag is the "softest" of the nonlinear problems to be considered, since its effects are relatively small at low velocities and thus can be easily overcome by slightly higher thrust levels. The plant equations of motion for the experiments of this section are:

$$\ddot{x} = -b\dot{x}\,|\dot{x}| + u \tag{4.9}$$

where b is the drag coefficient, and u is the applied control. Note that for low values of b, or simulations where the velocities remain low, the first term on the right hand side of (4.9) will contribute very little. One would thus expect the controller implemented by the network to be very similar to the canonical controller of Section 3.3.3 under these conditions.

Figure 4.2.1 shows the solo responses after the fiftieth iteration for two different values of the drag coefficient: b = 0.5 and b = 5.0. Although the water drag is ten times more significant in the latter case, the closed loop responses obtained by the network are virtually identical! This suggests that very different control strategies are employed, and in fact Figure 4.2.2 confirms this. The network has "learned" about the higher drag levels in the b = 5.0 experiment and provides more positive thrust over a longer period of time than

in the b = 0.5 experiment; in fact, while in the latter experiment the network must use negative thrust to brake to a stop at equilibrium, in the former experiment the network uses the higher drag levels to essentially coast to a stop. Again it must be emphasized that nothing in the payoff function explicitly tells the network about the larger drag levels, nor how to adjust to them; these different control laws have been constructed based upon the mutual interactions of the network, plant, and payoff function during the training phases.

Figures 4.2.3 and 4.2.4 show the network and resulting switching logic after the fiftieth iteration for b = 0.5, and Figures 4.2.5 and 4.2.6 show these diagrams for b = 0.5. Not only are the linear halves of the respective control laws different, the hidden neuron switching logic for the two drag levels is different as well. Although the magnitude of the contribution of each hidden neuron to the control is almost identical in the two experiments, the portion of the state space to which each responds is quite different for the two drag levels. The switching lines of the b = 0.5 experiment are very similar to those of the canonical run, which, as noted above, might have been expected given the relatively small contribution of the drag term in this case to the equation of motion. The switching lines of the b = 5.0 experiment are similar in their positioning and in the width of their linear regions to those of the lower drag experiment, but they are rotated somewhat from the b = 0.5 lines. In fact, it is this reorientation of the switching lines which creates the observed differences in the control profiles. As the state evolves from rest in the b = 5.0 experiment, only the output of neuron three begins to fall. The outputs of neurons four and five remain constant and even rise a bit as the velocity increases; in contrast, the output of all three hidden neurons in the b = 0.5 experiment begin to fall off rapidly as the plant velocity increases. By the time the velocity has reached its peak, the hidden neurons of the b = 0.5 network are almost half off, while those of the b = 5.0 network are still on as much as they were at the beginning of the simulation. Thus, while the control output by the b = 0.5 network drops rapidly as the state evolves, that output by the b = 5.0 network falls off much more slowly.

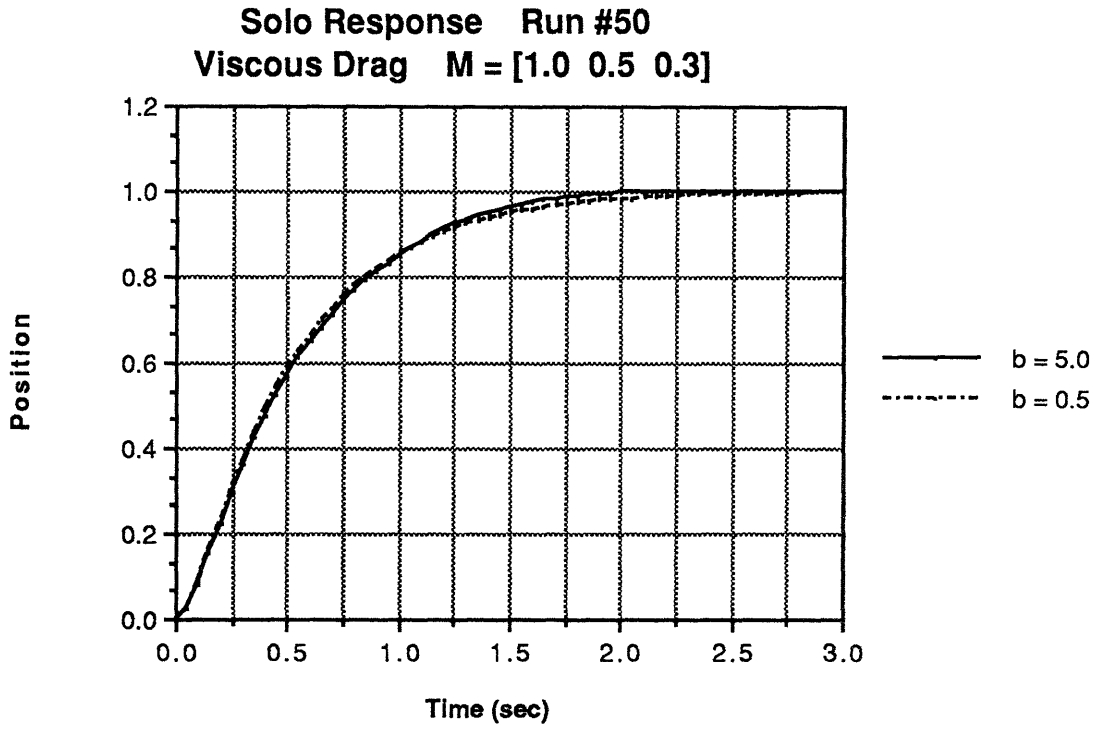

Figure 4.2.1: Solo responses after 50 iterations for high and low drag experiments

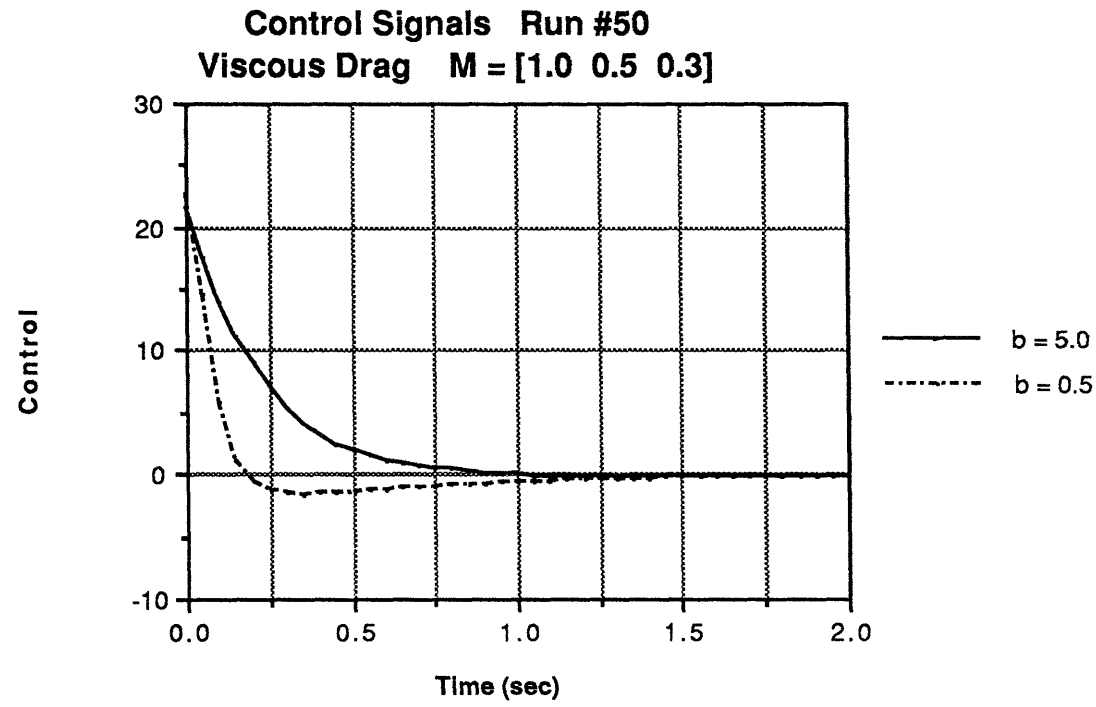

Figure 4.2.2: Control signals from network for high and low drag experiments

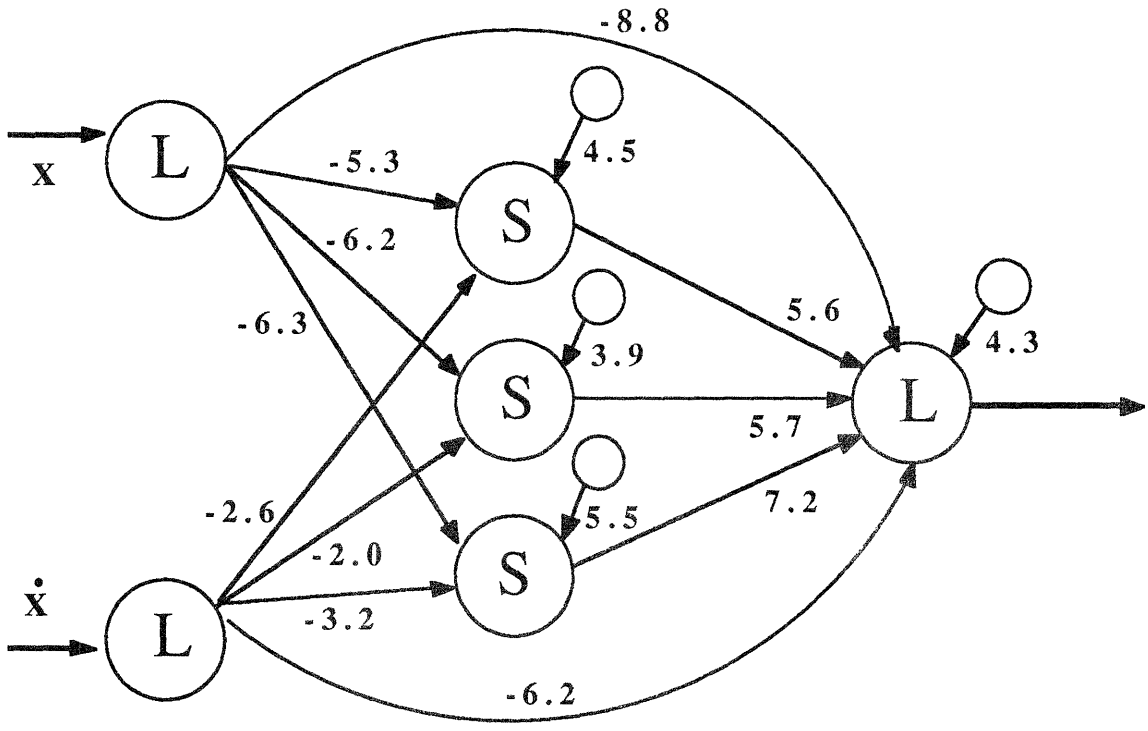

Figure 4.2.3: Network configuration after 50 iterations for low drag experiment

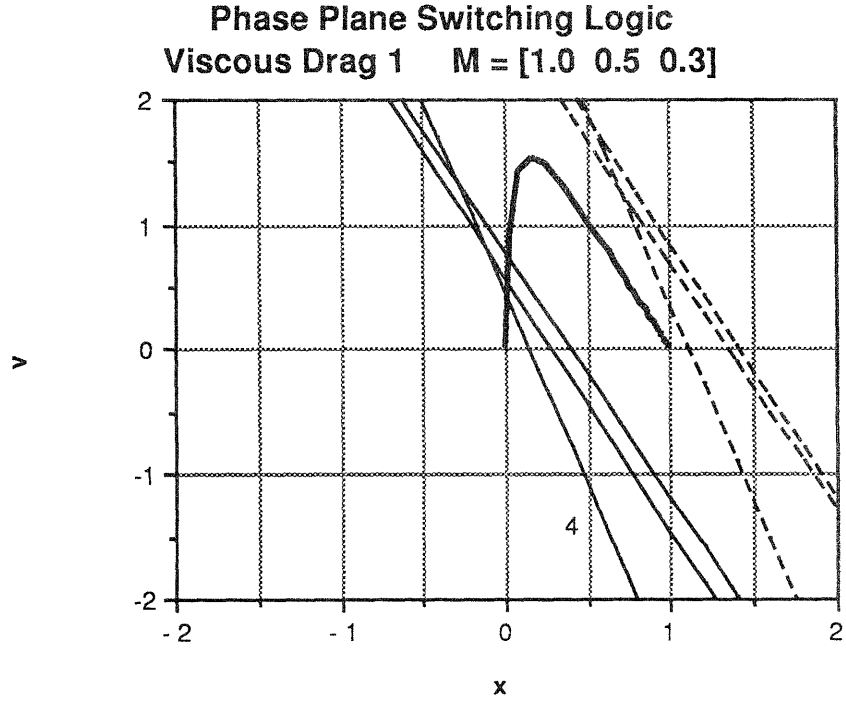

Figure 4.2.4: Switching logic implemented by the network of Figure 4.2.3

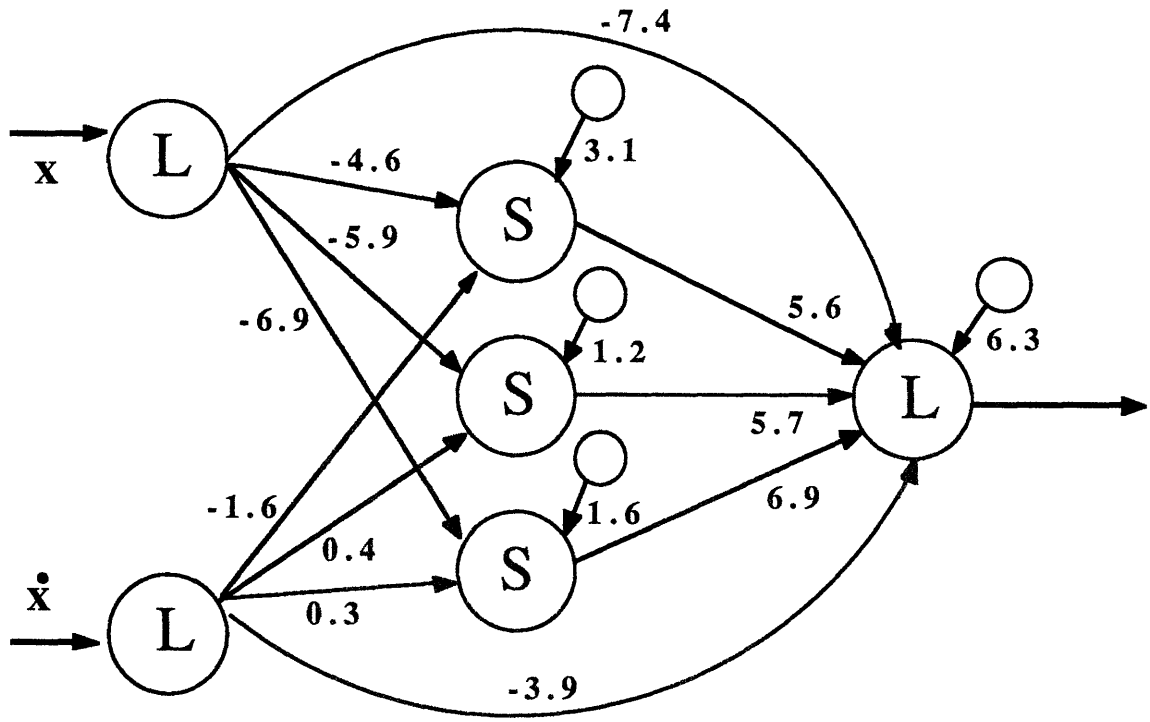

Figure 4.2.5: Network configuration after 50 iterations for high drag experiment

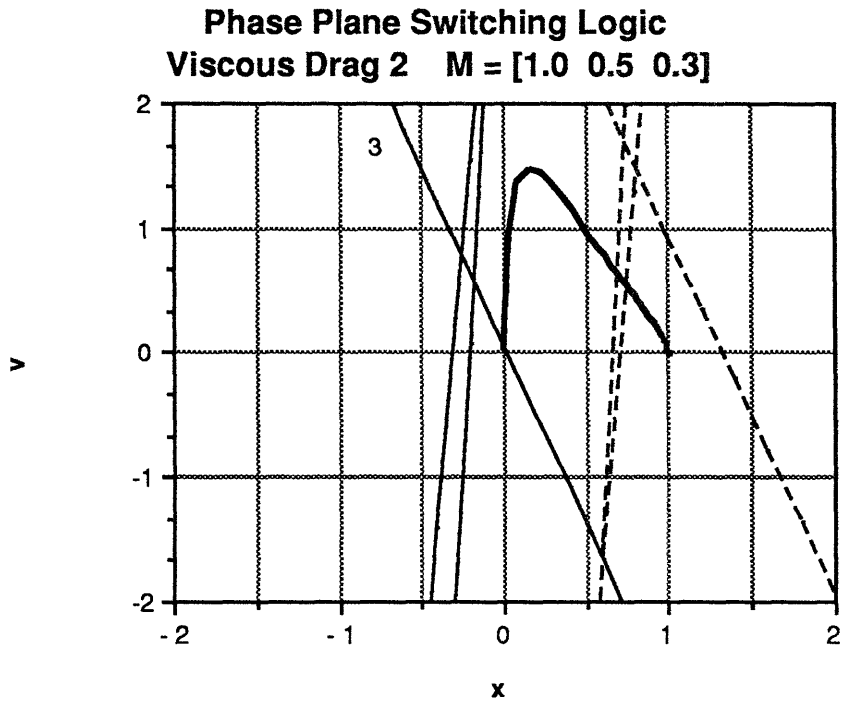

Figure 4.2.6: Switching logic implemented by the network of Figure 4.2.5

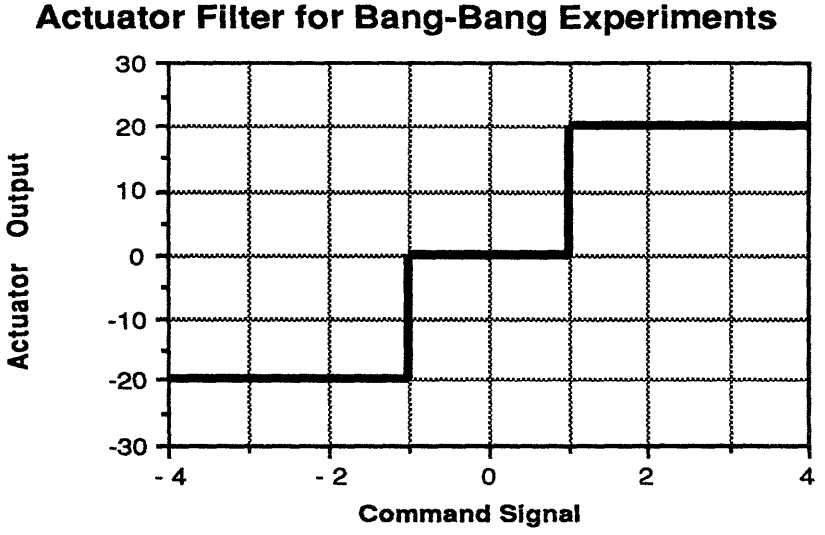

Figure 4.2.7: Actuation filter for the bang-bang controller experiments

4.2.2 Bang-Bang Actuation

For this experiment the NMC algorithm was again applied to the double integrator plant, but here the controls commanded by the network were first sent through the filter shown in Figure 4.2.7; the output of this filter is then applied to the plant as the control. Thus, commands from the network between -1.0 and 1.0 have no effect on the plant, commands larger than 1.0 cause +20.0 units of control to be applied, and commands smaller than -1.0 cause -20.0 units of control to be applied.

Note that this is actually quite a difficult problem for the network. In addition to developing a stabilizing switching logic, it must "learn" where the trigger levels of the actuator are. In initial experiments with the canonical weightings $\mathbf{m}^T = [1.0 \;\; 0.5 \;\; 0.3]$ it was found that the network would bring the plant to halt slightly short of or slightly beyond the desired equilibrium position. To prevent this, the position deviation weighting was increased to 2.5. Further, although limiting the magnitude of the control signal is not as critical in this experiment from a physical standpoint, a slight control weighting was used to keep the actual magnitude of the signals output by the network bounded. Thus, the weighting vector $\mathbf{m}^T = [2.5 \;\; 0.5 \;\; 0.02]$ was used for this section.

Figure 4.2.8 shows that the algorithm has again successfully devised a network which brings the plant to rest at exactly the desired equilibrium position. Note that the response is more "s"-shaped than the linear responses, and settles much more quickly than in the previous experiments. The controls applied to the plant, shown in Figure 4.2.9, tend to chatter quite a bit as the plant approaches equilibrium, but significantly there is no

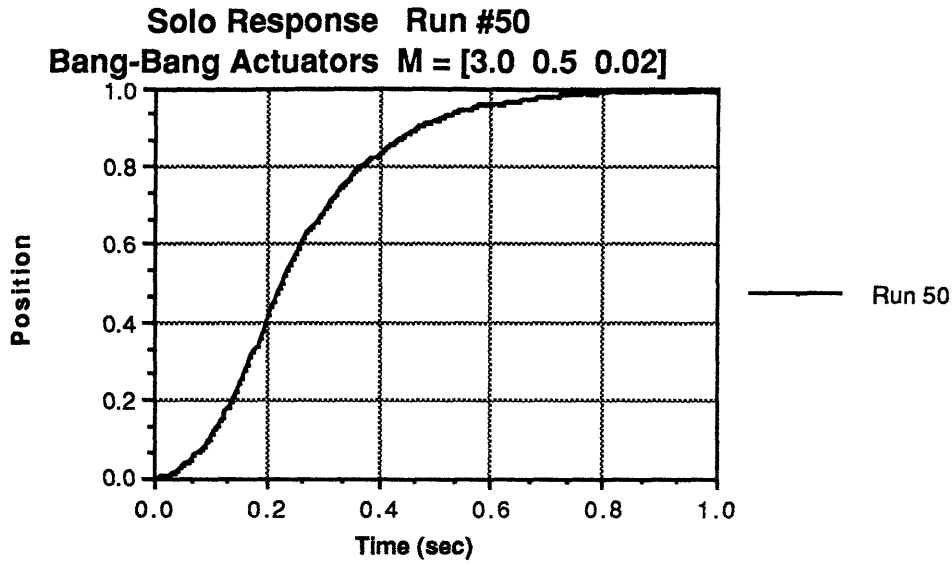

Figure 4.2.8: Solo responses after 50 iterations for the bang-bang experiment

limit cycling in the final position. Figure 4.2.10 shows the actual signals output by the network. Note that these also chatter, but this arises more from the impulsive changes in the velocity than any specific nonlinear action of the network itself. What is more interesting is that the chatter is about the -1 line; this is exactly where it must be in order to continue to trigger the actuator. This is a rather revealing example of the amount of learning done during the training periods; the network could not know about the actuator trigger levels except by experimentation with the dynamics during training.

Figure 4.2.11 shows the network which implements the controller. Showing the individual switching lines of the hidden neurons would not be of interest given the nature of these actuators; what is interesting is the actual switching logic which develops taking into account both the network and the actuation filter shown in Figure 4.2.7. There are two switching lines of interest here, the $+U_{max}$/off boundary and the $-U_{max}$/off boundary. These can be found by (numerically) solving:

$$1.0 = u_L(\mathbf{x}) + u_N(\mathbf{x}) \qquad (4.10)$$
$$-1.0 = u_L(\mathbf{x}) + u_N(\mathbf{x})$$

where $u_L(\mathbf{x})$ and $u_N(\mathbf{x})$ are the linear and nonlinear halves of the network control law, given by equations (3.2). The resulting switching lines together with the plant trajectory are shown in Figure 4.2.12. Notice that the equilibrium point lies, as it must to avoid limit cycling, exactly between the two boundary lines, and hence in that narrow region of state space where the actuators are off. These switching lines have a constant slope of about

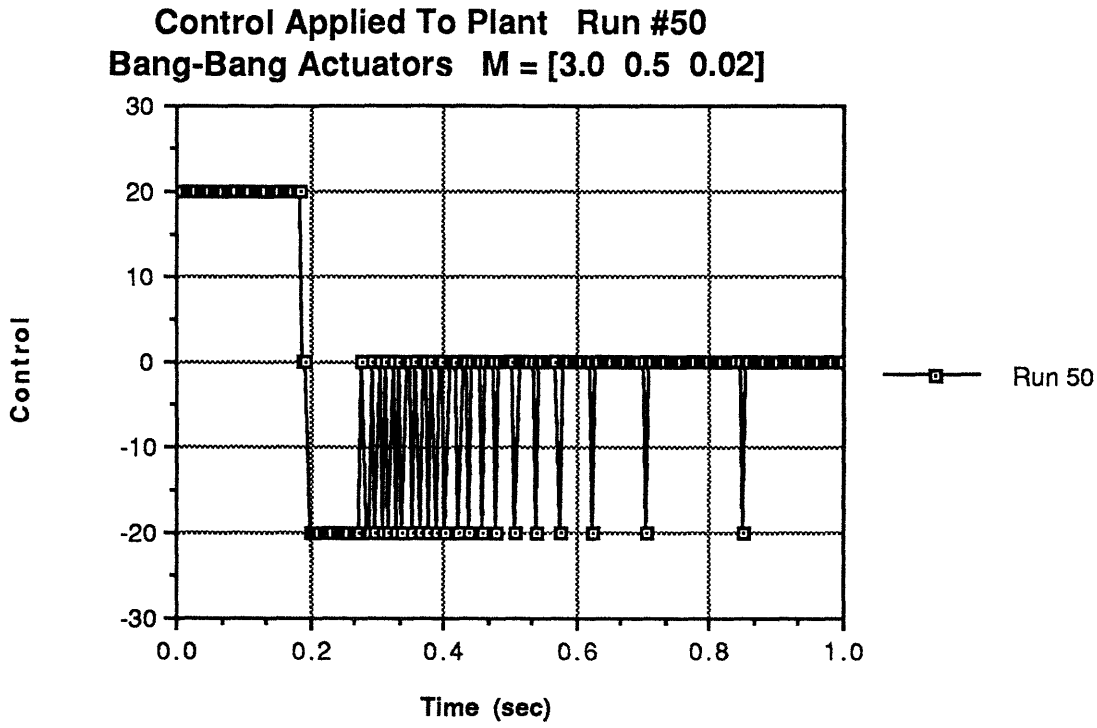

Figure 4.2.9: Control applied to the plant by the bang-bang actuators on the 50th iteration

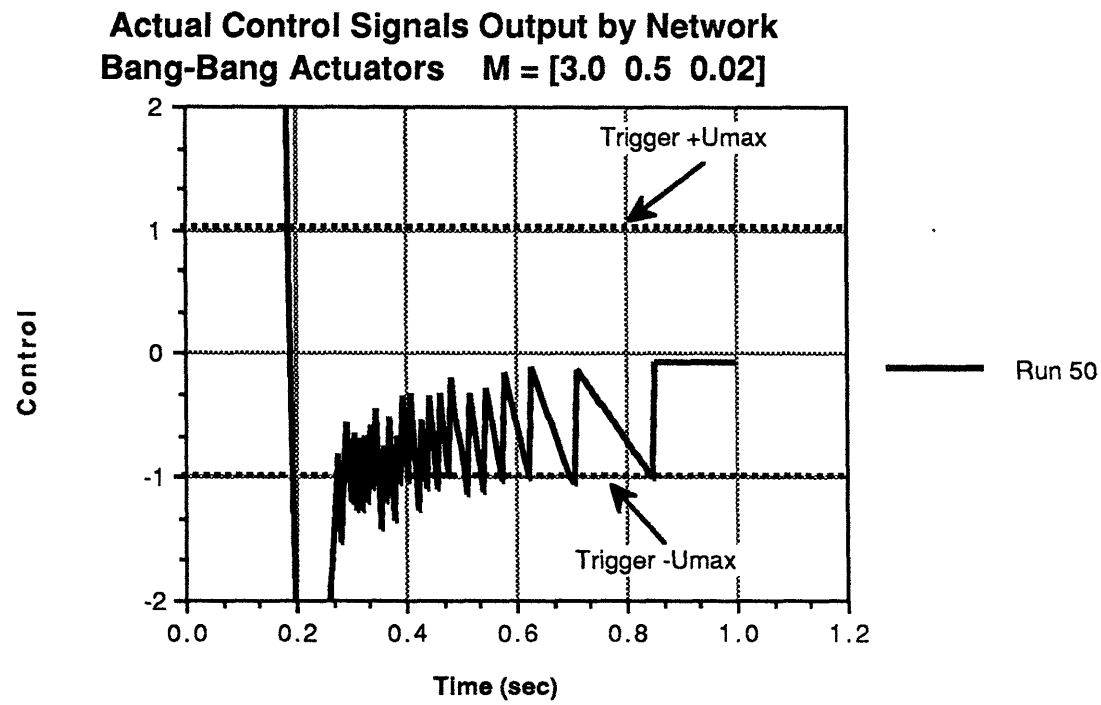

Figure 4.2.10: Actual commands issued by the network during the 50th iteration for the bang-bang actuation experiment.

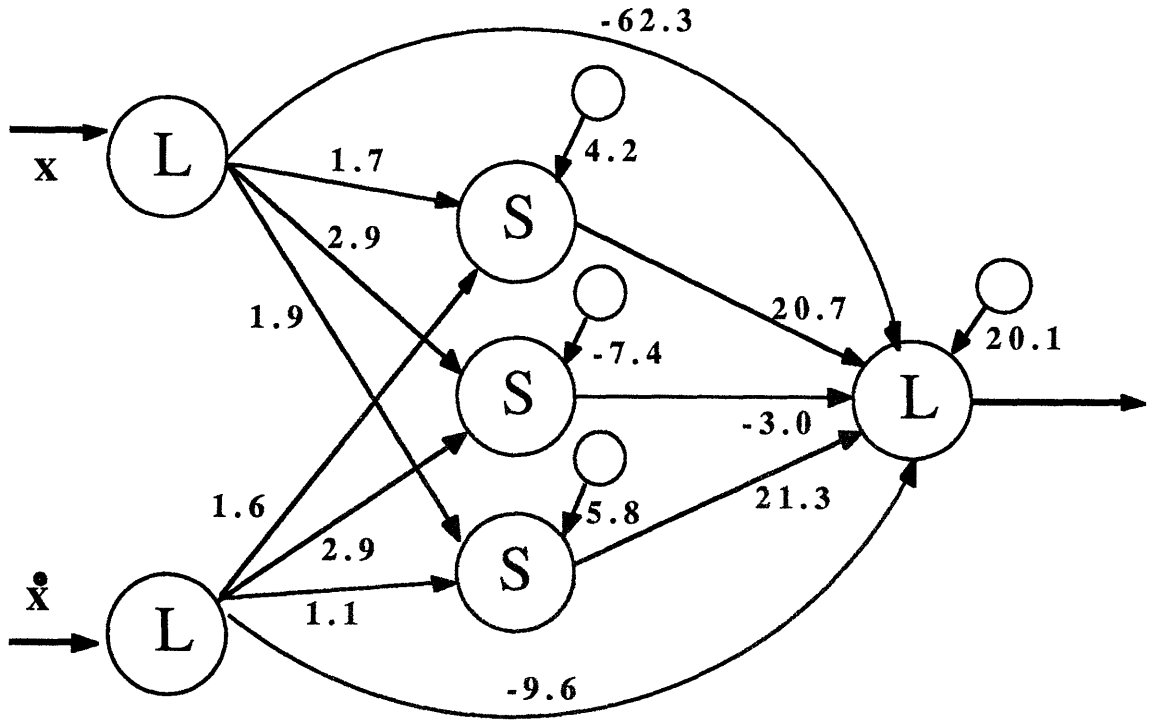

Figure 4.2.11: Network configuration after 50 iterations for the bang-bang experiment

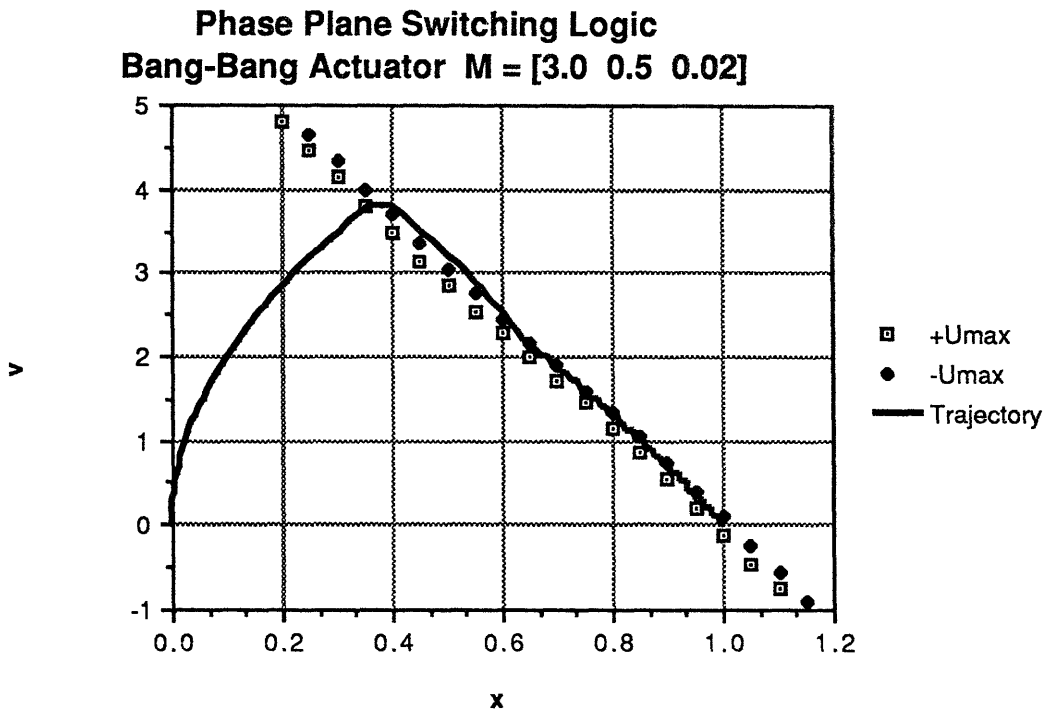

Figure 4.2.12: Switching logic taking into account the network dynamics of Figure 4.2.11 and the actuation filter of Figure 4.2.7

-6.25 and intercept the x axis so as to bracket tightly the x = 1.0 equilibrium point. The chatter is clearly seen in Figure 4.2.12; once the state has intercepted the switching lines, the plant is effectively guided along them to the desired final state, chattering down the $-U_{max}$ switching boundary.

4.2.3 MPOD Simulation

The final experiment with the NMC was conducted to demonstrate the feasibility of this algorithm for a system of "practical" complexity, and hence lay the foundations for a real time hardware implementation of this learning controller. The plant chosen for this experiment was a model of the pitch attitude dynamics of the SSL's Multimode Proximity Operations Device or MPOD, a teleoperator used to simulate satellite docking and servicing tasks in a neutral buoyancy environment.

The equations and physical parameters of MPOD have been exhaustively examined in several SSL theses and hence the derivations will not be detailed here. For the purposes of testing the NMC algorithm, the dynamic model is given by (Vyhnalek, 1985):

$$I_\theta \ddot{\theta} = - \, b|\dot{\theta}|\,\dot{\theta} - M_{B,\,max}\sin(\theta) + \tau \qquad (4.11)$$

where θ is the pitch angle, I_θ is the rotational moment of inertia about the pitch axis, b is the drag coefficient, $M_{B,max}$ is the maximum moment resulting from the offset of the center of buoyancy with respect to the center of gravity of the vehicle, and τ is the applied torque. Experimentally (Parrish, 1987) these values have been found to be approximately, $b/I \approx 2.5$, $M_{B,max}/I \approx 0.4$ rad/sec^2, and $I \approx 300$ kg-m^2. The control torque is delivered in 32 discrete levels, from -288 to +288 N-m. The actual torque delivered to MPOD as a function of the received commands, u, can be approximately modeled as a staircase function, given by:

$$\tau(u) = \begin{cases} 288 & \text{if } u > 16 \\ \text{int}(u)*18 & \text{if } |u| \le 16 \\ -288 & \text{if } u < -16 \end{cases} \qquad (4.12)$$

Two different regulator setpoints were simulated so as to explore the effects of the $\sin(\theta)$ nonlinearity at different pitch attitudes. The first setpoint was $\mathbf{x}_d^T = [1.0 \;\; 0.0]$,

where the pitch angular position is measured in radians; for this setpoint the sin(θ) term will be monotonically increasing as the plant evolves from rest to the equilibrium point. The second setpoint was chosen as $\mathbf{x}_d^T = [2.0\ \ 0.0]$, which ensures that the sin(θ) term is first increasing, then decreasing in its impact on the dynamics as the plant approaches equilibrium from rest.

Figure 4.2.13 shows the result after fifty iterations for the first setpoint. The response is very slightly underdamped, overshooting by 1.3%, and settles in about 2.6 seconds. Figure 4.2.14 shows the torques actually applied to MPOD (i.e. the result of sending the network's output through the staircase function of equation (4.12)); notice again the small amount of chatter required to maintain the setpoint. To offset the buoyancy moment at this first equilibrium point requires about 101 N-m of torque, which lies exactly between the +90 and +108 N-m torque levels which MPOD's actuators can provide; thus the network is switching the torque rapidly between these two levels to approximate the required intermediate level of thrust.

Figure 4.2.15 shows the results after fifty iterations for the larger angular maneuver. Here there is no initial overshoot, but again, as Figure 4.2.16 shows, the commanded torque tends to chatter in the steady state. Not shown in Figure 4.2.15 is a very slow overshoot of about 2% which accumulates after about 10 seconds; the controller has eliminated this by the 15 second mark. This slow overshoot, however, helps explain why the control chattering, between the +90 and +108 N-m levels as seen from Figure 4.2.16, is actually slightly below the +109 N-m required to maintain MPOD at the 2.0 radian pitch orientation; the controller is using the buoyancy moment to help slow the vehicle and counteract the building overshoot. In the steady state, after about 15 seconds, the chatter is reduced and the control is (mostly) constant at the +108 N-m level.

Figures 4.2.17 and 4.2.18 show the network and switching logic which have developed after fifty iterations for the first setpoint, while Figures 4.2.19 and 4.2.20 show these diagrams for the second setpoint. Notice that, for the first setpoint, the switching lines again resemble those of the canonical run or of the $b = 0.5$ drag experiment of Section 4.2.1, although the hidden neurons remain saturated on for significantly longer. In fact, the hidden neurons operate in the on saturation region until the vehicle is approximately 1/3 of the way to its equilibrium position, contributing +33 units and hence ensuring that the actuators are operating at maximum output. As the state approaches equilibrium, these gradually begin to reduce to about 50% output each in the steady state, providing the commands necessary to counteract the resulting buoyancy moment.

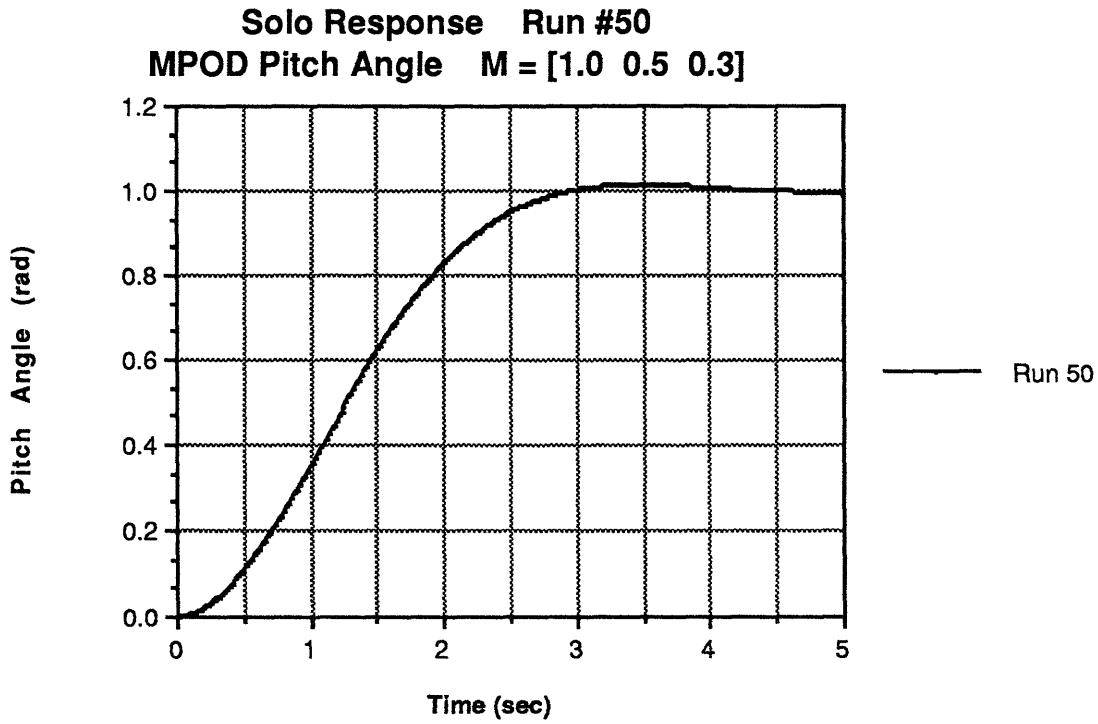

Figure 4.2.13: Solo response after 50 iterations for 1 rad MPOD pitch maneuver

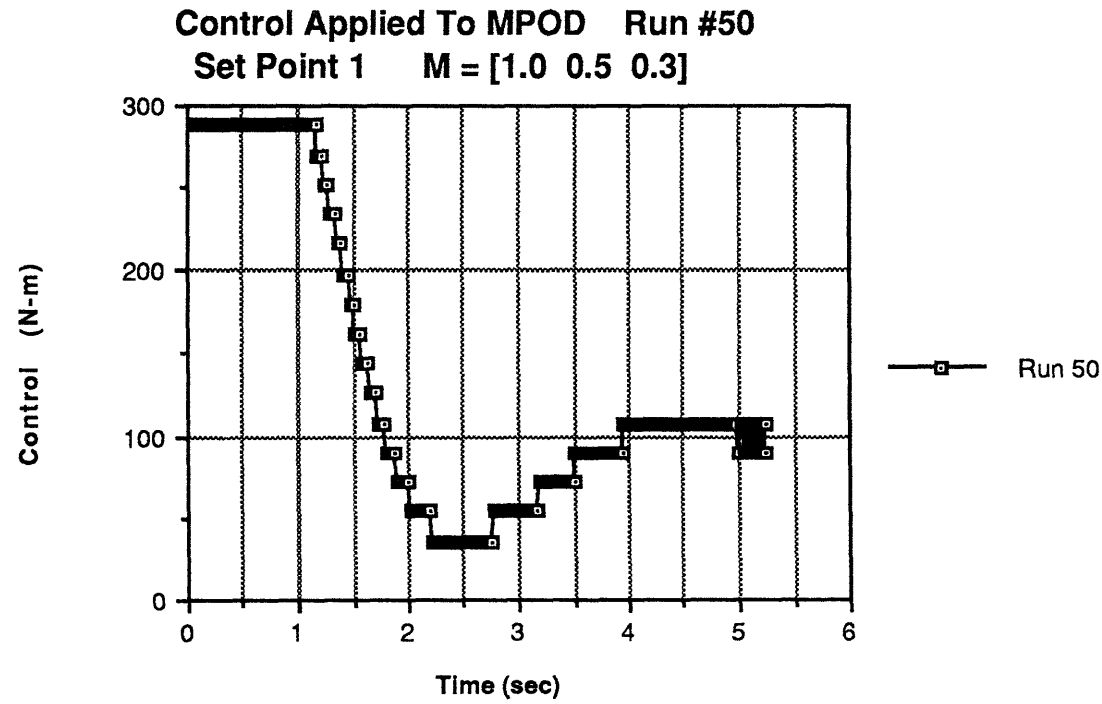

Figure 4.2.14: Controls applied by MPOD's thrusters during 1 rad pitch manuever on the 50th iteration

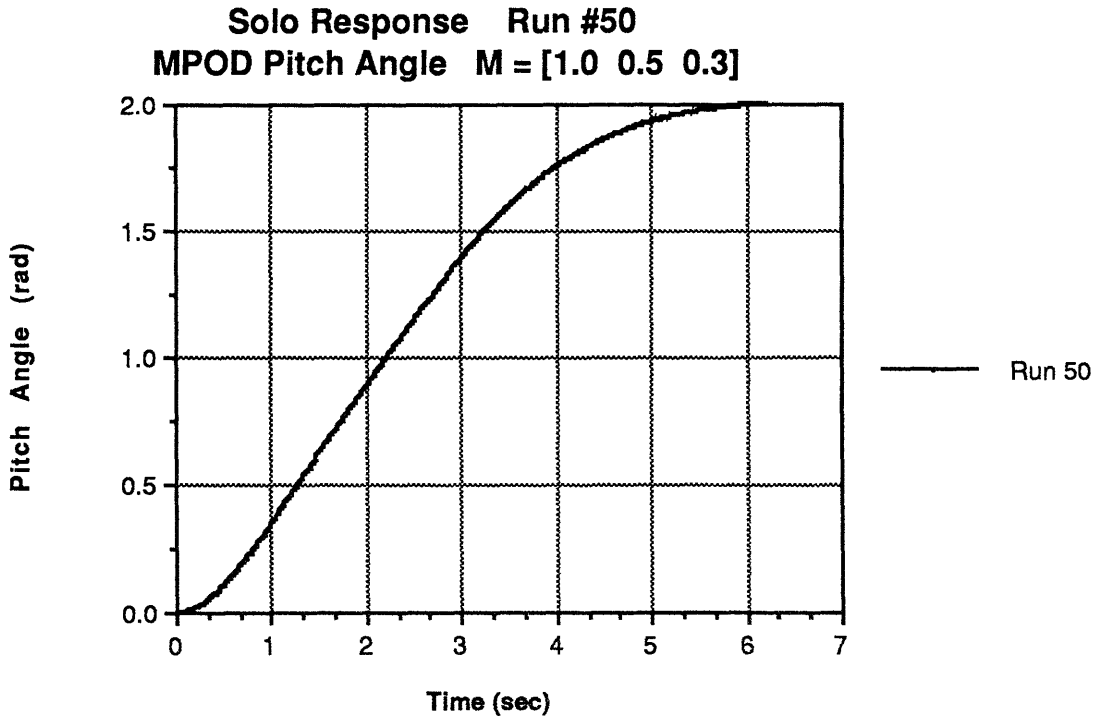

Figure 4.2.15: Solo response after 50 iterations for 2 rad MPOD pitch maneuver

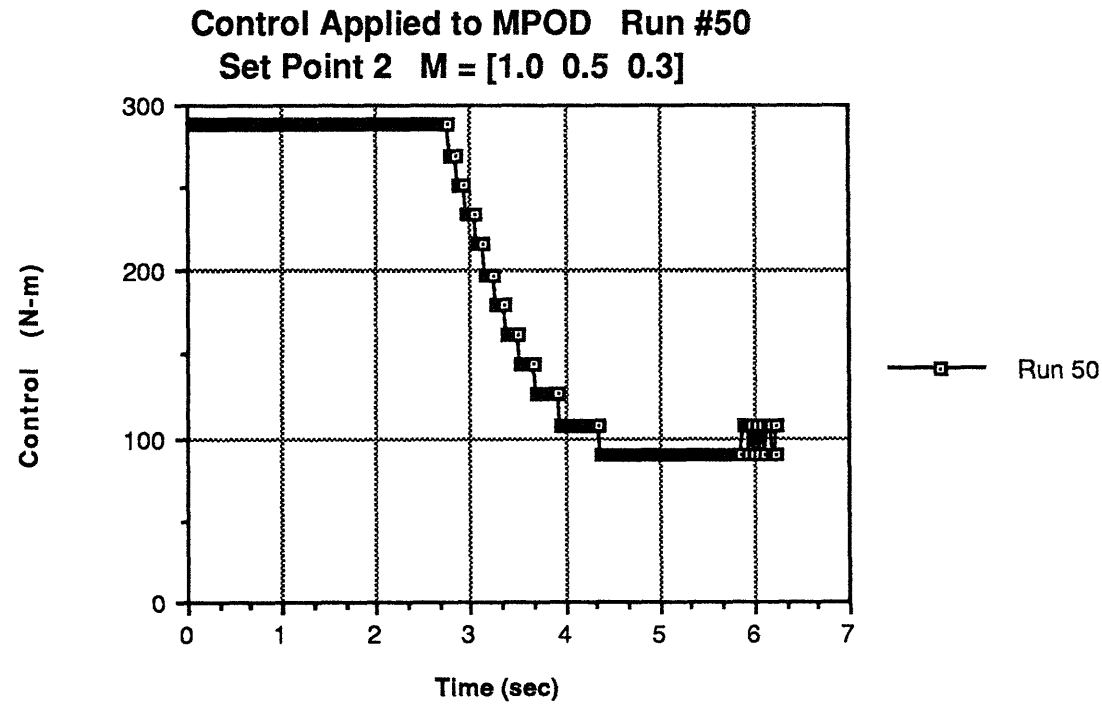

Figure 4.2.16: Controls applied by MPOD's thrusters during 2 rad pitch manuever on the 50th iteration

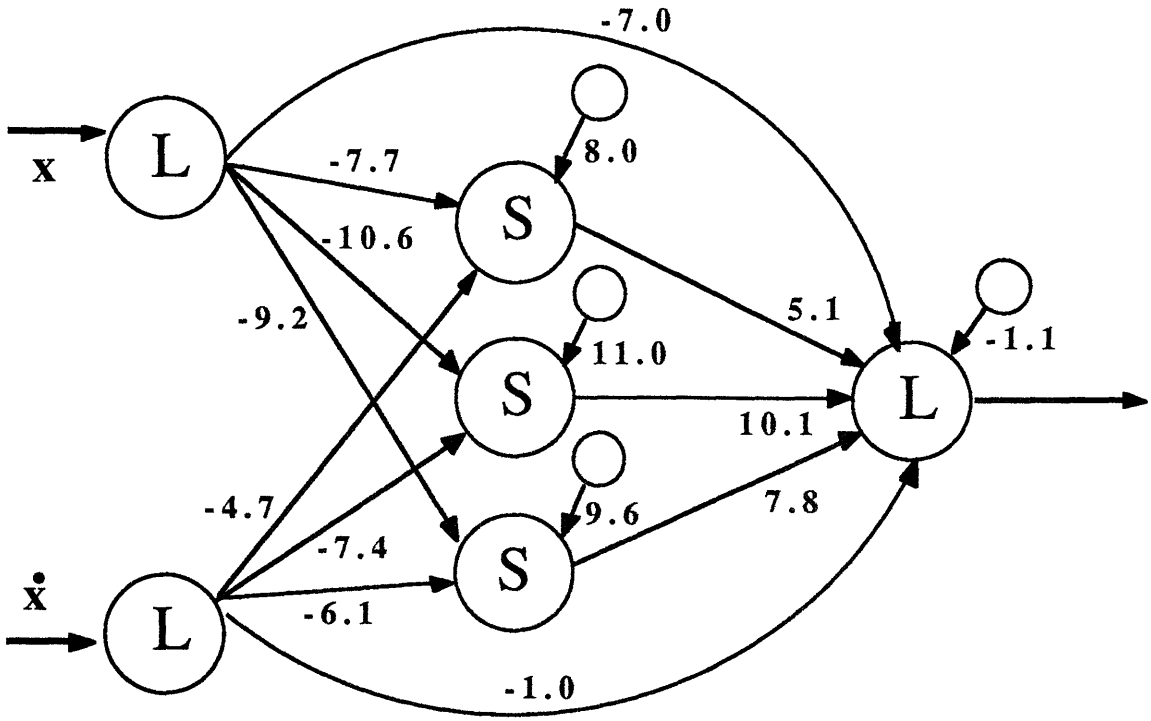

Figure 4.2.17: Network developed after 50 iterations for 1 rad MPOD pitch maneuver

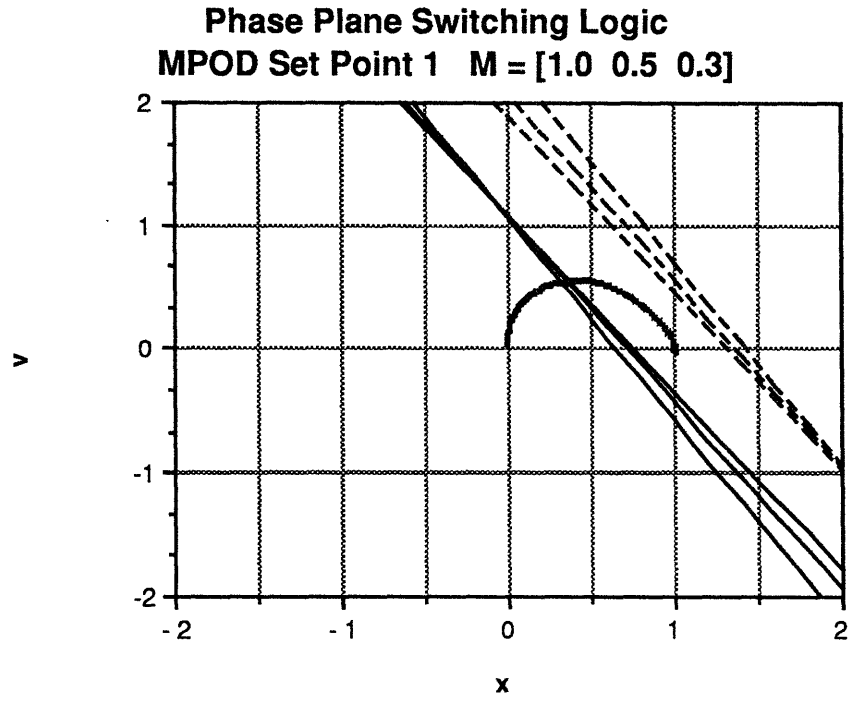

Figure 4.2.18: Switching logic implemented by network of Figure 4.2.17

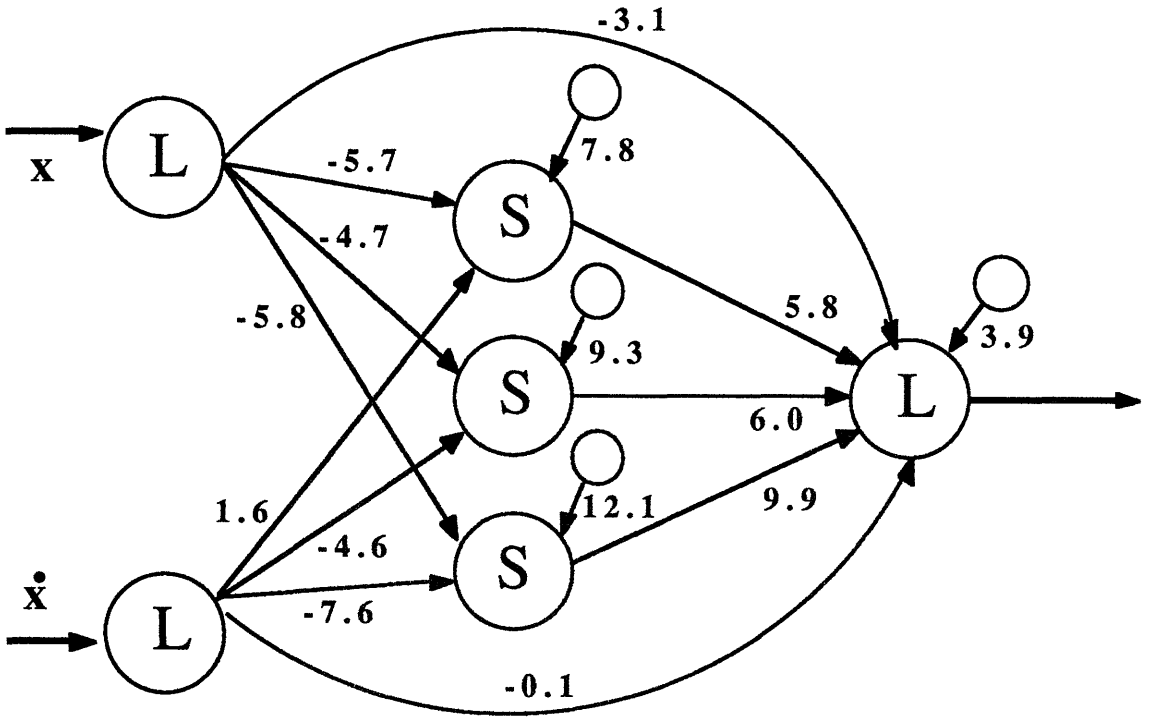

Figure 4.2.19: Network developed after 50 iterations for 2 rad MPOD pitch maneuver

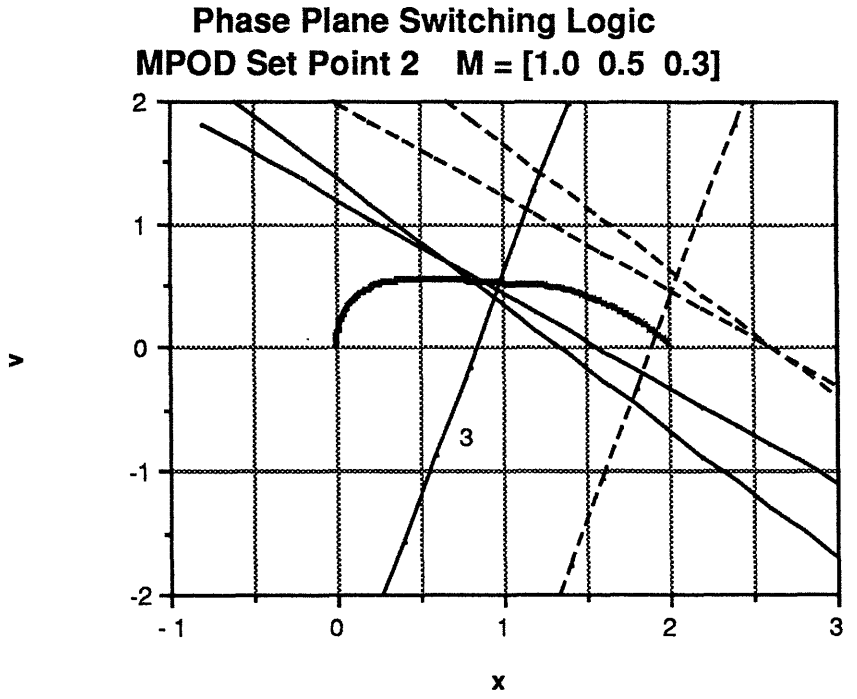

Figure 4.2.20: Switching logic implemented by network of Figure 4.2.19

The switching curves for the second setpoint are more revealing. These curves are still arranged so that the hidden neurons begin the simulation in the on saturation region, again ensuring the actuators are saturated at +288 N m. The controller implemented by the network brings MPOD to a velocity of 0.5 rad/sec and essentially coasts at that velocity until about half the desired pitch angle is obtained. At this point the trajectory intersects the critical point formed by the intersection of hidden neuron three's switching line with those of hidden neurons four and five at $\mathbf{x}_c^T = [0.95 \;\; 0.5]$. As the state crosses this point, each hidden neuron begins to shut off: neuron three eventually comes fully off, while neurons four and five remain on 50% in the steady state. This drop in hidden neuron output, coupled with the linear feedback terms, accounts for the large, smooth drop in the control from the +288 N m level to the +90 N m level seen in Figure 4.2.15. Significantly, this drop is occurring as the contribution of the sin(θ) term is reaching its maximum and beginning to decrease, i.e. when the pitch angle approaches and exceeds 1.57 rad.

Thus the MPOD simulations, like those involving bang-bang actuators, viscous drag, and noisy sensors, has led to the development of control laws which are *nonlinear* over the range of states experienced in the simulations. This is in marked contrast to the linear systems experiments above, where most of the control laws developed were completely linear during the simulation. It is important to note that all of these control schemes have been implemented on the same network and further that, regardless of the character of the control law, every neuron participates in its formulation. While the algorithm seems to "prefer" linear controllers, probably arising from the fact that the teaching stimulus is linear in the states, it will use its nonlinear degrees of freedom when it must to account for the plant dynamics or operating conditions.

4.3 NMC Failure Modes

Several cases have been observed in which the neuromorphic control algorithm fails to converge. These failures all occurred during the training period; there have been no observed cases where the algorithm stabilized during the training phase and yet was unstable during the solo runs. Each training failure became manifest when the response of the system continually exceeded the absolute bounds on performance imposed by the trainer (i.e. $x(t) \geq 10x_d$, or $u(t) \geq 5u_{ult}$. A particularly common form of failure occurred when the network weights grew so rapidly during training that the maximum control bound

imposed by the trainer was exceeded for any value (including zero) of the plant states. The simulator would thus become locked into a cycle in which a reset was signalled at every time step; this would continue until manually interrupted. This section presents a summary and brief analysis of each of the situations which evoked this behavior.

The first class of observed failures points out one of the fundamental limitations of the NMC algorithm: at each point in time that a synaptic weight change is initiated, the *sign* of the payoff function must correctly reflect the required change in the magnitude of the control signal; e.g. if δ is positive, this indicates that a more strongly positive control is required to force the plant state to the desired equilibrium, and vice-versa. It is this requirement which prevents a (weighted) sum squared error metric from being used as the payoff function.

Notice that this limitation forces the designer to know the sign of the impact of the control on the states of the plant. If this sign is inverted (for example, if the control gain in the double integrator plant examined in Chapter 3 were negative), the algorithm will become unstable; in this case the "criticism" provided by the trainer is essentially telling the network to adjust the control in exactly the wrong direction. A more subtle manifestation of the same problem occurs when the plant is nonminimum phase; i.e. when the output one wishes to regulate contains the negative of one or more of the plant states. In this situation, the transients of the output tend to first move further from their desired equilibrium positions before settling. This behavior clearly confuses the algorithm which *occasionally* becomes unstable with these plants. Interestingly, there were several times when the network did converge to a stabilizing controller for a nonminimum phase plant, but since these results were not repeatable with different initial conditions, the algorithm was judged unstable in this case, and hence the results are not included in this thesis.

No solution is known for the problem posed to the NMC by nonminimum phase plants; indeed this type of plant can cause problems for many conventional adaptive control schemes. A possible solution to the sensitivity of the algorithm to the sign of the plant control gain would be to make the payoff function itself adaptive; that is, allow certain changes to occur to the structure of the critical signal provided by the trainer based upon observations of the correlation of the criticism and some metric of controller performance improvement. This "metacritical" level is almost, but not quite, a method of introducing into the NMC algorithm a more conventional on-line determination of the sensitivity derivatives of the plant and controller. "Almost" so, because this metacriticism can be theoretically quite crude, containing only sign information, in order to address the problems noted above.

The second form of observed failures occurred when the magnitude of the payoff signal grew "too large". In the original derivation of the back propagation algorithm, the magnitude of the output layer error signal never grew larger than ±1.0. In the NMC algorithm, depending upon the values of the coefficients in equation (3.7) or the magnitude of the desired final states, the payoff signal can grow substantially larger than 1.0. In fact, the state weighted payoff function algorithm was observed to be unstable for values of $\mathbf{m}^T = [7.0\ 0.5\ 0.3]$, with $\mathbf{x_d}^T = [1.0\ 0.0]$, and also for $\mathbf{m}^T = [1.0\ 0.5\ 0.3]$, with $\mathbf{x_d}^T = [5.0\ 0.0]$. The resulting large $\delta(t)$ produces large weight changes at each synaptic update step, which seems to incite unstable oscillations in the network weights. Tight bounds must be thus maintained on δ if the algorithm is to converge.

Clearly this particular failure mode places a limitation on the utility of the algorithm. It is easy to restrict the payoff function weights, $\mathbf{m}$, but one would like to be able to command arbitrary setpoints, $\mathbf{x_d}$, and still be guaranteed a stable learning process. One could, of course, train the network on a smaller setpoint, such as $\mathbf{x_d}^T = [1.0\ 0.0]$, then introduce a bias into the sensors (with the trainer off!), but this could cause serious problems for nonlinear plants, which may exhibit greatly different dynamics as the magnitude of their states grow. One possible way to rectify this problem in the NMC algorithm might be to *scale* each contribution of the state deviations in the payoff signal to their desired final values. A more general formulation for the payoff signal, then, might be:

$$\delta(t) = M_x\left(\frac{x_d - x(t)}{x_d}\right) + M_{\dot{x}}\left(\frac{\dot{x}_d - \dot{x}(t)}{\dot{x}_d}\right) - M_u \mathrm{sgn}(u)\left|\frac{u}{u_{ult}}\right|^n \qquad (4.13)$$

If any of the desired states is zero (for example $\dot{x}_d$ in a position regulator), the denominator of that term's contribution to the payoff should be replaced with unity. In fact, a preliminary evaluation of this new payoff function was performed for $\mathbf{x_d}^T = [7.5\ 0.0]$, and the algorithm successfully converged to a stabilizing controller.

The final form of instability witnessed is related to the above problem of restricting the absolute magnitude of the payoff signal. This last problem arises whenever the control signals commanded during the training phase begin to exceed substantially u_{ult}. Such an event can occur in two situations: when using (for the double integrator plant, at least) a state weighted payoff function in which the velocity weighting is zero, i.e. $\mathbf{m}^T = [M_x\ 0.0\ M_u]$; and when the amount of control required to even stabilize an unstable plant is approximately the same, or larger than, u_{ult}, such as with the plant:

$$G(s) = \frac{1}{(s+5)(s-5)} \tag{4.14}$$

when $u_{ult} = 16.0$. In both these cases, during its first training run the network finds itself in a configuration where the plant has drifted very far from the desired equilibrium; in order to regain the equilibrium configuration, the algorithm tries to increase the synaptic weights and hence command larger controls. If these controls begin to exceed u_{ult} (as they must just to stabilize the plant in (4.14)) their impact on the payoff function becomes quite large and the unstable oscillations in the network weights discussed above occur.

In all cases observed, increasing u_{ult} or decreasing M_u removed this instability; this is in agreement with the results of Sections 3.3.1 and 3.3.2 where it was shown that the algorithm produced stabilizing controllers even in the absence of explicit control weightings. In general, however, this change in the payoff function will lead to (perhaps undesirable) changes in the shape of the closed loop response; this is an excessive price to pay to avoid a source of instability which will arise only during the first training iteration. It has already been demonstrated that the network tends to arrive at the "form" of the control law it implements after only one or two iterations, then slowly begins to tune this control law, lowering u_{max} to the optimal level, in the sense of minimizing $\delta(t)$. This suggests that an "annealing" approach to the contribution of the control to the payoff function might be successful; i.e. start the algorithm with little or no control weighting in the payoff function, then slowly increase this weighting to the desired level as the number of training phases increases. Equivalently, one could do the inverse with u_{ult}.

Chapter 5: Conclusions

5.1 Observations and Caveats

This thesis has demonstrated the feasibility of using one of the new neural network architectures as the basis for controllers which learn to regulate dynamic systems. This is *not* to suggest that what has been presented constitutes a fully developed adaptive control algorithm. The distinction between adaptive and learning controllers, as noted in the Introduction, is subtle but important; the fundamental differences in these concepts alone would rule out any immediate practical applications of the above results. One could hardly allow an airplane to crash a few times before the neural controller invented a stabilizing control scheme. But then, one would not expect a novice human pilot to perform much better without training! A possible method for learning controllers in general, and the NMC in particular, to be used as viable control strategies for complex systems would be to allow the network to perform its first training runs in a *simulated* environment, as is done with human pilots for air- and spacecraft. The resulting trained network could then be brought on-line into the real-time control situation. The control law it has developed based upon its interactions with the simulator should be sufficient to at least ensure stability, if not superb performance, provided the training model has been sufficiently accurate. The NMC could then fine tune its control law, and hence increase its performance, as a result of the real-time interactions with the physical process.

As adaptive schemes go, the NMC algorithm is rather slow, requiring usually between twenty and one hundred seconds of (simulated) training time until it converges to the final form of its control law. Compare this with, for example, an adaptive sliding mode controller or similar parameter estimating adaptive algorithm, in which the controller parameters can be ascertained (given a sufficiently rich trajectory) in only a few tenths of a second; although, to be fair to the NMC, parameter estimating adaptive architectures are heavily dependant upon the assumed structure of the plant and the form of a stabilizing control law, while the neural controller must essentially learn anew the principles of feedback control each time it is run. The observed rate of convergence is probably wholly attributable to the (somewhat arbitrary) topology and training parameters chosen for analysis: no attempt was made to increase the performance in this respect, and there is

evidence, even among the training parameters chosen for analysis, that faster learning is possible without jeopardizing stability.

In fact, the NMC has accomplished a great deal given its extremely simple architecture. Other experiments with back propagation networks have involved thousands of neurons, and tens to hundreds of thousands of synaptic connections; the NMC examined here is implemented with (for second order plants) just ten neurons and fifteen synapses, and yet learns very complicated (and very useful!) mappings. Sejnowski and Rosenberg's (1987) simulation which learned to read English text aloud took almost one week on a DEC VAX 11/780 to train; by contrast, the NMC algorithm completes a set of fifty iterations in about two hours on an IBM PC/AT (depending upon the complexity of the plant equations of motion).

Further, there is nothing special in the topology of the network or in the structure of the training algorithm to make the networks examined especially suitable for control purposes; the NMC algorithm and network topology examined above might equally well be trained to implement XOR, or any other desired mapping. This lack of structure, or perhaps the ability to create their own structure, is one of the most exciting aspects of neural network designs in general. Coupled with the observed ability of the network to determine which of a set of input stimuli were pertinent for controlling the plant (q.v. Section 3.5.4), this opens the possibility for more advanced adaptive control architectures.

However, nothing has been mathematically *proven* regarding the performance of the NMC algorithm. Any justification lies solely in analogy to the back propagation proofs, supported by about one hundred experiments which only *demonstrate* the stability of the algorithm for a few plants, about still fewer operating points, and in a *simulated* environment. Moreover, it has been shown that certain choices of the algorithmic parameters will result in instability and, even though many of these can be addressed as discussed in Section 4.3, there are probably many other, yet undiscovered, combinations of plants and algorithmic parameters which will also result in unstable operation. To have complete confidence that the results presented above are not special cases of a fundamentally unstable algorithm, rigorous proofs would need to be developed which can provide justification for, and predict limitations on, the stability of the learning process. Given the highly nonlinear nature of the network dynamics as a whole, such proofs are likely to be difficult at best.

Finally, the results detailed above only demonstrate the ability of a neural network to *regulate* a dynamic system, and that only after it has traversed essentially the same path through state space several times. It is a more difficult problem to design a similar

algorithm in which a neural network learns to behave as a true *controller*, capable of guiding the plant along an arbitrary desired trajectory; a trajectory which it may or may not have previously experienced during its training phases. The goal of this thesis has been to demonstrate the feasibility of using neural networks for control applications; the results reported are considered encouraging enough to begin intensive research into these more sophisticated neuromorphic controllers, and some preliminary ideas along these lines are presented in the next section.

5.2 Recommendations for Future Research

Most of these recommendations on the subject of neural controllers center around resolving the above noted problems which have emerged in the course of this preliminary analysis of this topic. Clearly the first priority should be the construction of correct proofs which demonstrate when and why such an algorithm will successfully converge to stabilizing control laws. A good initial step in this direction would be to devise such a proof for just the double integrator plant and second order network configuration described in Chapter 2 and 3, and demonstrate the observed instabilities mathematically. Such a proof might help establish a theoretical framework in which the relation between the form of the payoff function, the actual plant dynamics, and the range of plant initial conditions used in training could be reliably quantified and used to predict the final configuration of the synaptic weightings and the resulting neural switching logic.

The next priority should be to use the results of this thesis to construct a true neuromorphic *control* network, instead of one which functions solely as a state regulator. As with the neuromorphic regulator, a fully trained neuromorphic controller should be able to, in the presence of a time invariant plant, follow any desired trajectory without any further intervention of the trainer: the trained network should thus implement a perfect closed loop tracking system. Construction of such a network would require some changes in the topology and training system developed above, but these extensions should be straightforward. As a first cut, for example, one could add two input layer neurons which specify the desired final state for the network; this would probably also require more hidden layer neurons. In the same vein, it would be interesting to explore the concept of a neural *identifier*, in which the network is presented with a (discrete) time varying input, u_k, and required to produce a set of n outputs, $\mathbf{x}$, which satisfy the relation:

$$\mathbf{x}_{k+1} = g(\mathbf{x}_k, u_k) \tag{5.X}$$

Here the error signal at each of the n output nodes would be the deviation of that state from the value it should have given the dynamic relations, g(•), and the control time history.

It has already been noted that the back propagation algorithm upon which the NMC is based is only one of a plethora of neurally inspired algorithms. Given the serious limitations inherent in the back propagation information processing paradigm, there is adequate motivation to consider some of these other algorithms, or at least attempt to incorporate their features into the structure of back propagation. In particular, it is extremely difficult to have a back propagation network "understand" time as an external variable. Since these networks employ neurons whose outputs develop "instantaneously" as functions solely of their current net input, there is no dependence in the network on the previous neural states; the back propagation nets examined in this thesis can hence be realistically used only for static pattern matching. This presents a serious drawback from a control standpoint. One possible way to introduce a temporal aspect into the operation of back propagation, and hence into the NMC algorithm, would be to allow feedback connections in the network, both between layers and between individual neurons within a layer. In this way, every time the net is "pulsed" in response to a new set of inputs, the values which develop at the output layer neurons will be functions not only of the current inputs, but also of the past network state. Rummelhart and Hinton have already made some preliminary investigations into this extension to back propagation, and in fact have demonstrated that such *recurrent* nets demonstrate the same convergence properties as the previously detailed networks.

Many of the sections in Chapters 3 and 4 leave one wondering "what if...?"; which is precisely the same question which inspired the results cited in those chapters. The NMC algorithm detailed above is hence itself worthy of further study without any modifications. What is the optimal number of hidden layer neurons; how will the algorithm function for even higher order linear and nonlinear plants; to what extent does the range of states encountered in the training sequence determine the control laws developed by the network; can this methodology be extended to multiple input and output plants; these are but some of the questions which arose in the preparation of this thesis. Clearly, time constraints prevented these from being analyzed in further detail.

Finally, despite the warnings issued in the previous section, a hardware implementation of the NMC should be effected as soon as possible on one of the SSL's underwater teleoperators. It is necessary to ensure that the observed simulation results, encouraging though they are, can in fact be reproduced in a physical system. It is no

accident that the pitch attitude dynamics of MPOD was chosen for analysis at the end of Chapter 4; at this stage, MPOD is the vehicle in the laboratory most suited for an experimental control algorithm of this type.

In fact, the necessary software has been written, debugged, and installed on MPOD's control station RECS, the Reconfigurable Experimental Control Station. Only a serious hardware failure in MPOD's pressure distribution system midway through the Spring 1988 academic term prevented the results of this implementation from being cited herein. Complete repairs to the vehicle have been almost completed as this thesis is being prepared, and it is expected that the NMC will have its first hardware test sometime during June 1988. Important factors to assess in these hardware tests will be the ability of the algorithm to design networks even in the face of substantial time delays in the downlink path, and sensor noise corrupting not only the network inputs, but also the training signal itself. Once the algorithm has proven itself effective in real time, it would be valuable to compare the performance of a "pretrained" network, i.e. a network which has first been trained on the MPOD simulator used in this thesis and then installed in the actual vehicle, with a network which acquires all of its training while on-line.

The ultimate aim of the future research described in this section would be to address the criticism, posed in Section 5.1, of the suitability of these architectures for actual implementation; the use of neural networks as elements of robust, unstructured adaptive controllers is a very tantalizing goal, but much work yet needs to be done to establish the feasibility of this idea.

5.3 Summary and Conclusions

This thesis has proposed and evaluated several extensions to the classical back propagation methodology for training neural networks. These extensions consist of three fundamental modifications. First, the paradigm of an omniscient teacher imposing a strict functional relationship between the input and output layers has been relaxed and replaced by a *critic*. This critic, observes the interaction of the network with its surrounding environment and generates a signal which represents a qualitative evaluation of how well the network is producing the desired effects in this environment. The criticism, or *payoff*, signal is applied at the output terminals of the network and back propagated through each layer in the conventional manner, providing the training stimulus for each synaptic update phase. The second change is the use of *linear response neurons* in both the input and output layers so as to encode the fullest possible dynamic range of impinging

environmental stimuli and the signals exported by the network in response. Finally, the training set for the network is not specified through a pretabulated mapping table, but rather arises as the result of the real time interaction of the network with a dynamic system, whose states evolve in time driven by the outputs of the network.

The motivation behind these modifications has been to evaluate the ability of back propagation networks to learn to regulate dynamic systems about a specific desired plant state. For this purpose it is assumed that neither the network, nor the critic, have any *a priori* knowledge of the plant nor of the form of a stabilizing control law. In this context, the above ideas have been incorporated into the *neuromorphic controller*, a methodology for the *learning control* of dynamic systems. The distinction drawn between learning controllers and conventional adaptive controller is primarily based upon how stability is achieved. While training, learning controllers may experiment with the plant dynamics, perhaps driving the system unstable, and require outside intervention (a *trainer*) to "catch" the plant, reset the system, and restart the training. The measure of stability for a learning controller is whether, after a finite period of training, it has developed a control law which is asymptotically stable to the desired equilibrium state, without any further intervention of the trainer. Adaptive controllers, by contrast, while they may tune in real time the parameters of their control law, must *never* allow the plant to become unstable. Learning control is hence a less restrictive methodology than adaptive control.

The neuromorphic control algorithm has been demonstrated to produce networks which implement control laws which bring the plant exponentially to the desired equilibrium for a range of second order, and one third order, linear and nonlinear plants, driven by both linear and nonlinear actuators. For the second order plants and the network topology most extensively studied, the control law implemented by the network has fifteen tuneable parameters, corresponding to the fifteen synaptic weights. Depending upon the exact values of these parameters, the resulting controller can be either completely linear, or composed of as combination of linear and nonlinear terms; although even in this latter case the linear terms will tend to dominate when the plant is far from equilibrium. The precise values of the synaptic weights determine the regions of state space to which the nonlinear network elements, e.g. the hidden layer neurons, respond most strongly. The hidden neurons can thus be considered as adaptive "feature detectors" on the state space, which strategically introduce control signals when the plant approaches certain configurations, in such a way as to improve some metric of system performance.

Analysis has shown that this metric is minimization of the magnitude of the payoff signal, pointwise in time. The "optimal" solution for the controller would thus be to adjust its weights so that $\delta(t) = 0.0$ for all time. This minimization requires the network and its

trainer to perform real time tradeoffs among the parameters of the payoff function. Moreover, this tradeoff must be conducted in conjunction with the evolution of the plant dynamics during the training periods, since the interrelations of the plant state variables which appear in the payoff function are unknown *a priori.* One could not predict the final form of the control law based solely on inspection of the payoff function; at least rudimentary aspects of the plant dynamics must be "discovered" by the network. Hence, guided by the criticism provided in the payoff function, and based upon its experiences with the plant dynamics during the training periods, the neuromorphic controller *develops its own unique control strategy.* This is very different than a conventional back propagation result in which the network can learn only those mappings presented by the trainer; with the NMC, the pupil can actually surpass its teacher.

However, precisely because the control strategy is generated in association with the evolution of the plant dynamics, the actual control laws developed depend upon exactly how much of the state space has been experienced by the network while training. This suggests that the algorithm presented is rather sensitive to the range of plant initial conditions used during the training period, and raises questions about the degree to which the training set has been "sufficiently rich" in order that the resulting control law be valid on the entire state space, not merely on those states visited while training. Further, some dependence of the control law upon the initial conditions of the network has also been noted. For a given payoff function, plant, and set of initial plant conditions used during training, the control laws generated in different trials of the algorithm will be virtually identical when the network starts with small, random synaptic weights; under the same training conditions, when the network starts with large, grossly dissimilar initial synaptic weights, the algorithm will converge to a different control law. However, in each of these cases certain features of the "shape" of the closed loop plant response which results remain very similar, suggesting that the control problem as posed to the network is underconstrained.

Quite often, the network was observed to implement a control law which was completely linear over the range of states experienced moving the plant from its initial condition to the desired equilibrium. In these networks, characteristic of neural network solutions in general, control authority was distributed throughout all the neurons of the network. However, when required in order to accomplish the desired minimization of the payoff signal, the network can introduce nonlinear action into the control law in such a way as to, for example, reduce overshoots, overcome plant nonlinearities, make effective use of bang-bang actuation, and suppress sensor noise. In many of these cases it can be argued that the network has developed a primitive, *crude plant model* based upon its experiences

during the training phase, and thus *feedsforward* a certain, predetermined, amount of control, in addition to the linear feedback terms, so as to force the plant more swiftly to equilibrium. This was evident, for instance, in the high inertia and high drag plants where the hidden layer neurons were saturated on, contributing large amount of positive control, when the plant started from rest, but which shut rapidly off as the plant approached equilibrium. Further, there is evidence to suggest that the network can distinguish between relevant and irrelevant environmental stimuli by detecting correlations between activity in the input layer and the evolution of the payoff signal. The network actually attempts to *remove irrelevant stimuli* from the network, and hence reduce or eliminate their impact on the control law.

The very essence of neurocomputing, as evidenced by the preliminary research emerging from this nascent field, seems to underscore the *nonalgorithmic* nature of the decision logic implemented by neural networks. In fact, for sufficiently complex networks, it may prove impossible to "unscramble" the decision scheme embedded in the synaptic weights, one could only verify that the network performs as desired. This feature may the price one must pay for the plasticity these networks; the same plasticity which would, for example, allow the network topology most heavily analyzed in this thesis to equally well be trained to implement an XOR gate, or any other two to one mapping. Fortunately, for the particularly simple network topologies analyzed in this thesis, it was possible to analyze and verify the logic employed by the network, however, this would not, in general, hold true for more complicated neural controllers. This may be somewhat aggravating to the scientist, who is left without a precise model with which to predict the behavior of the system, but it is intuitively correct in light of our own experiences with biological neurocomputers, e.g. our own brains. One could hardly explain, either mathematically or verbally, how one coordinates the actions of myriads of muscles and tendons with the sensory feedback required to, for example, hit a baseball. The fact that we cannot explain exactly how we do it, however, does not negate the fact that we have hit the ball.

It is this fundamental intuition which lies at the heart of this thesis. Biological systems are phenomenal examples of robust, adaptive MIMO controllers, yet it is not likely that any of these systems contain explicit, structured models of the dynamics which govern their movement. Although it can be (correctly) argued that many of the traits exhibited by living creatures are "hardwired" into their physical structure, there is ample evidence for learned sensory motor behavior, such as walking in humans. A child learns to walk, probably not by least squares tuning of regulator gains, but by qualitatively associating certain kinds of sensory feedback with unpleasant events, such as falling, and other kinds

of feedback with pleasant events, such as walking to his or her mother. By trial and error, typically falling quite often in the process, the child makes the correct mappings of "sensors" to "actuation" which results in consistently stable perambulation. Learning control using neuromorphic controllers is intended to be a crude analog of this process, and the model analyzed above presents just a hint of what this concept, coupled with the new neural architectures, might allow in the way of providing machines with the same functionality. It is hoped that the results of this thesis, and the recent advances in neurocomputing in general, are considered encouraging enough to spur further research into this fascinating topic.

References

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J., "A Learning Algorithm for Boltzmann Machines," Cognitive Science, vol. 9, pp. 147-169, 1985.

Adrian, R. H., "The Nerve Impulse," *Carolina Biology Readers, #67*, J. J. Head, ed., Carolina Biological Supply Co., Burlington, NC, 1980.

Alspector, Joshua, and Allen, Robert B., "A Neuromorphic VLSI Learning System," in *Adv. Res. in VLSI: Proc. Stanford Conf.*, Stanford CA, 1987.

Anderson, David E., "Supervisory Control Algorithms for Telerobotic Space Structure Assembly," SM Thesis, MIT Dept. of Aero. and Astro., May 1988.

Barto, Andrew G., and Sutton, Richard S., "Landmark Learning: an Illustration of Associative Search," *Bio. Cybern.*, vol. 42, pp. 1-8, 1981.

Barto, Andrew G., Anderson, Charles W., and Sutton, Richard S., "Synthesis of Nonlinear Control Surfaces by a Layered Associative Search Network," *Bio. Cybern.*, vol. 43, pp. 175-185, 1982.

Barto, Andrew G., Sutton, Richard S., and Brouwer, Peter S., "Associative Search Network: a Reinforcement Learning Associative Memory," *Bio. Cybern.*, vol. 40, pp. 201-211, 1981.

Barto, Andrew G., Sutton, Richard. S., and Anderson, Charles W., "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," *IEEE Trans. Sys., Man, and Cybern.*, vol. SMC-13, pp. 834-846, 1983.

Cajal, S. R., *Histology*, Wood, Baltimore, 1933.

Carpenter, Gail, and Grossberg, Steven, "A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine," *Computer Vision, Graphics, and Image Processing*, vol. 37, pp. 54-115, 1987.

Castellucci, V., and Kandel, E. R., "Presynaptic Facilitation as a Mechanism for Behavioral Sensitization in *Aplysia*," *Science*, vol. 194, pp. 1176-1178, 1976.

Cohen, Michael, and Grossberg, Steven, "Absolute Stability of Global Pattern Formation and Parallel Memory Storage by Competitive Neural Networks," *IEEE Trans. Sys., Man, and Cybern.*, vol. SMC-13, pp. 288-308, 1983.

Fu, K. S., "Learning Control Systems," in *Computer and Information Sciences*, Julius Tou and Richard Wilcox, ed., Spartan Books, Washington, pp. 318-343, 1964.

Gorman, R. Paul, and Sejnowski, Terrence J., "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets," *Neural Networks*, vol. 1, pp. 75-89, 1988.

Gray, E. G., "The Synapse," *Carolina Biology Readers, #35*, J. J. Head, ed., Carolina Biological Supply Co., Burlington, NC, 1977.

Grossberg, Steven, "Adaptive Pattern Classification and Universal Recoding, I: Parallel Development and Coding of Neural Feature Detectors," *Biological Cybernetics*, vol. 23, pp. 121-134, 1976(a).

Grossberg, Steven, "Adaptive Pattern Classification and Universal Recoding, II: Feedback, Expectation, Olfaction, and Illusions," *Biological Cybernetics*, vol. 23, pp. 187-202, 1976(b).

Grossberg, Steven, *Studies of Mind and Brain*, Reidel Press, Boston, 1982.

Grossberg, Steven, "Competitive Learning: From Interactive Activation to Adaptive Resonance," *Cognitive Science*, vol. 11, pp. 23-63, 1987.

Grossberg, Steven, "Nonlinear Neural Networks: Principles, Mechanisms, and Architectures," *Neural Networks*, vol. 1, pp. 17-61, 1988

Grossberg, Steven, and Kuperstein, Michael, *Neural Dynamics of Adaptive Sensory-Motor Control: Ballistic Eye Movements*, Elsevier, Amsterdam, 1986.

Guez, Allon, Eilbert, James, and Kam, Moshe, "Neuromorphic Architectures for Fast Adaptive Robot Control," *IEEE Proc. Intl. Conf. on Robotics and Automation*, Philadelphia PA, I, pp. 145-149, April, 1988.

Hebb, D. O., *The Organization of Behavior*, New York, Wiley, 1949.

Hestenes, David, "How the Brain Works: the Next Great Scientific Revolution," Third Workshop Max Entropy and Bayesian Meth. Appl. Stat., Univ. of Wyoming, Aug, 1983.

Hinton, G. E., and Sejnowski, T. J., "Learning and Relearning in Boltzmann Machines," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol 1, David Rummelhart and James L. McClelland, ed., MIT Press, Cambridge, pp. 282-317, 1986.

Hopfield, J. J., "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *Proc. Natl. Acad. Aci., USA*, vol. 79, pp. 2554-2558, 1982.

Hopfield, J. J., "Neurons with Graded Response Have Collective Computational Properties Like Those of Two-State Neurons," *Proc. Natl. Acad. Aci., USA*, vol. 81, pp. 3088-3092, 1984.

Hopfield, J. J., and Tank, D. W., "'Neural' Computation of Decisions in Optimization Problems," *Bio. Cybern.*, vol. 52, pp. 141-152, 1985.

Jackel, L. D., Howard, R. E., Graf, H. P., Straughn, B., and Denker, J. S., "Artificial Neural Networks for Computing," J. Vac. Sci. Technol. B, vol. 4, no. 1, pp. 61-63, 1986.

Kandel, Eric R., and Schwartz, J. H., *Principles of Neural Science*, Elsevier, New York, 1985.

Kawato, M., Furukawa, Kazunori, and Suzuki, R., "A Hierarchical Neural-Network Model for Control and Learning of Voluntary Movement," *Bio. Cybern.*, vol. 57, pp. 169-185, 1987.

Kirkpatrick, S., Gelatt, C. D. Jr., and Vecchi, M. P., "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671-680, 1983.

Kohonen, Teuvo, *Self-Organization and Associative Memory*, Springer-Verlag, Berlin, 1984.

Kuperstein, Michael, "Generalized Neural Model for Adaptive Sensory Motor Control of Single Postures," *Proc. IEEE Intl. Conf. Robotics and Automation*, Philadelphia, PA, vol. I, pp. 140-143, April 1988.

Kurtzman, Clifford R., "Time and Resource Constrained Scheduling, with Applications to Space Station Planning," Ph.D. Thesis, MIT Dept. of Aero. and Astro., Feb 1988.

Landahl, H. D., McCulloch, W. S., and Pitts, Walter, "A Statistical Consequence of the Logical Calculus of Nervous Nets," *Bull. Math. Biophys.*, vol. 5, p. 135-137, 1943.

Lorentz, G. G., "Metric Entropy, Widths, and Superposition of Functions," *Amer. Math. Monthly*, vol. 69, pp. 469-485, 1962.

Lorentz, G. G., "The 13th Problem of Hilbert," in *Proc. Symp. Pure Math.*, Feliz E. Browder, ed., vol. 28, pp. 419-430, 1976.

McCulloch, Warren S., and Pitts, Walter, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bull. Math. Biophys.*, vol. 5, p. 1115-133, 1943.

Miller, Rene H., Minsky, Marvin L., and Smith, David B. S., *Space Applications of Automation, Robotics, and Machine Intelligence Systems (ARAMIS)*, NASA CR-162079, Aug. 1982.

Minsky, Marvin, and Papert, Seymour, *Perceptrons*, Cambridge, MIT Press, 1969.

Parrish, Jospeh C., "Trajectory Control of Free-Flying Space and Underwater Vehicles," SM Thesis, MIT Dept. of Aero. and Astro., Aug. 1987.

Pavlov, Ivan P., *Lectures on Conditioned Reflexes*, New York, International Publishers, 1928.

Rohrs, Charles E., Valavani, Lena, Athans, Michael, and Stein, gunter, "Robustness of Adaptive Control Aglorithms in the Presence of Unmodeled Dynamics," *Proc. 21st IEEE Conf. on Decision and Control*, Ft. Lauderdale Dec. 1982.

Rosenblatt, Frank, *Principles of Neurodynamics*, Washington, Spartan Books, 1962.

Rummelhart, D. E., Hinton, G. E., and McClelland, J. L., "A General Framework for Parallel Distributed Processing," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol 1, David E. Rummelhart and James L. McClelland, ed., MIT Press, Cambridge, pp. 45-76, 1986(a).

Rummelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing: Explorations in*

the Microstructure of Cognition, vol 1, David Rummelhart and James L. McClelland, ed., MIT Press, Cambridge, pp. 282-317, 1986(b).

Sejnowski, Terrence J., and Rosenberg, Charles R., "Parallel Networks That Learn to Pronounce English Text," *Complex Systems*, vol. 1, pp. 145-168, 1987.

Showlater, Barton, SM Thesis, MIT Dept. of Aero. and Astro., in preparation.

Sprecher, D. A., "On the Structure of Continuous Functions of Several Variables," *Trans. Amer. Math. Soc.*, vol. 115, pp. 340-355, 1964.

Stevens, Charles F., *Neurophysiology, A Primer*, Wiley and Sons, New York, 1966.

Stevens, Charles F., "The Neuron," *Scientific American*, September, 1979.

Tank, David W., and Hopfield, John J., "Simple 'Neural' Optimization Networks: An A/D Converter, Signal Decision Circuit, and a Linear Programming Circuit," *IEEE Trans. Circ. and Sys.*, vol CAS-33, pp. 533-541, 1986.

Tarrant, Janice M., "Attitude Control and Human Factors Issues in the Maneuvering of an Underwater Space Simulation Vehicle," SM Thesis, MIT Dept. of Aero. and Astro., Aug. 1987.

Viggh, Herbert E. M., "Artificial Intelligence Applications in the Teleoperated Robotic Assembly of the EASE Space Structure," SM Thesis, MIT Dept. of Aero. and Astro., Feb 1988.

Vyhnalek, Gary G., "A Digital Control System for an Underwater Space Simulation Vehicle Using Rate Gyro Feedback," SM Thesis, MIT Dept. of Aero. and Astro., May 1985.

Widrow, B., and Hoff, M. E., "Adaptive Switching Circuits," *Inst. Rad. Eng., Western Electric Show and Conventions, Convention Record,* IV, p. 96-104, 1960.

Widrow, Bernard, and Smith, Fred W., "Pattern-Recognizing Control Systems," in *Computer and Information Sciences*, Julius Tou and Richard Wilcox, ed., Spartan Books, Washington, pp. 288-317, 1964.

Widrow, Bernard, Gupta, Narendra, and Maitra, Sidhartha, "Punish/Reward: Learning with a Critic in Adaptive Threshold Systems," *IEEE Trans. Sys., Man, and Cybern.*, vol SMC-3, pp. 455-465, 1973.

Appendix A: The NMC Software

In the interest of providing support for future research at the SSL into neural networks, the simulator package was constructed to support as many different neural models and training algorithms as possible. The logical choice for such an implementation was an object oriented programming style, which allows the software developer to concentrate on more or less abstract manipulations of high level data types (objects) without concern for how each object accomplishes the requested actions.

For the NMC simulation, two object classes exist called *net* and *neuron*. *Nets* can be given three instructions: they can be told to update themselves (which is tantamount to ensuring that each neuron which comprises the net is updated), to submit to a teaching iteration, to display their current state in graphical form, and finally, they can be told to dump their state to an (already open) data file. *Neurons* can similarly be commanded to update their current state, and to make an axonic connection to another neuron.

Listing A.1 shows the C header file which creates the templates for each object. The structure Network consists of a list of neurons, a vector representing the values of the current output layer neurons, and pointers to three functions: a Teaching function, a Displaying function, and a Debugging function. For this thesis, the Displaying function was never written--space for it is reserved in the *net* template, however, to allow for the time when "real time" (hopefully color) displays of the evolution of the network become viable. The structure Neuron consists of: pointers to a neural activation function, the derivative of this activation function, and an output function; storage locations which indicate the current values of the input, output, and error signal, as well as a fourth unused location for future expansion; a tag which identifies the type of neuron (input layer, output layer, hidden layer, or bias); and finally, pointers to the heads of two lists of structures called Synapses, one for the axonic connections and one for the dendritic.

The pattern of interconnections made by each neuron is summarized in lists of Synapses which are "daisy chained" together. A null pointer for the head of either the axonic or dendritic list indicates that the neuron has none of the specified connections. Otherwise, the pointer in the structure Neuron holds the address of the first synapse in the respective list. Each synaptic structure consists of two storage locations, one specifying the current synaptic strength and one provided for future expansion, as well as a pointer the the neuron at the other end of the synapse. The final element of the structure is a pointer to the next synapse in the list; this will be null if the current synapse is at the end of the list. Notice that, taken together, the axon-dendrite lists for all neurons in the network will be redundant, since the axons of one neuron are the dendrites of another. It is, however,

conceptually and computationally easier to deal with this redundancy than to implement a more compact model.

Different "flavors" of objects can be created by filling the (initially empty) function pointers in the object templates. The numeric storage locations are used during computation and do not generally need to be filled by the user. Each of the functions associated with an object should conform to the function protocol for that object. These will be detailed below.

One flavor of network object was created. The protocol for all the functions associated with network objects is to take a single argument which is a pointer to the network. The *backprop* flavored network has the subroutine **BackProp** as its teaching function, and the subroutine **Dump** as its debugging function. **BackProp(network)** implements the back propagation algorithm detailed in Chapter 2.0. **Dump(network)** simply copies the current state of each neuron in the network to a formatted data file. The source code for both of these subroutines is listed in Listing A.2.

Several flavors of neurons were created. The protocol for the functions associated with neuron objects is that each should take a single floating point argument: the activation and derivative functions take the total neural input **net** , while the output function takes the current neuron activation value **output** as its argument. The *linear input* flavor has the activation function **Forced(net)**, the derivative function **DerivLinear(net)**, and the output function **Identity(output)**. **Forced(net)** sets the current state of the ith input neuron equal to the value contained in the ith position of the external array **input_vector.** The *hidden* flavored neuron has **Sigmoidal(net)** as its activation function, **DerivSigmoidal(net)** as its derivative function, and **Identity(output)** as its output function. The *bias* flavored neurons have the activation function **AlwaysOn(net)**, the derivative function **DerivSigmoidal(net)** (although the derivative function is never needed for *bias* neurons in *backprop* flavored networks), and again **Identity(output)** is the output function. Finally, *linear output* flavored neurons have the activation function **Linear(net)**, the derivative function **DerivLinear(net)**, and the (as usual) **Identity(output)** is the output function. The C source code for all these activation functions is listed in Listing A.3.

New *neuron* flavors are created using the subroutine **MakeNeuron**. This subroutine takes as arguments the type of neuron being defined (input, output, bias, or hidden), as well as pointers to the activation function, derivative function, and output functions. **MakeNeuron** creates a new neuron object, fills the template with the specified information, initializes the storage locations and synaptic lists, then returns to the caller the address of the new object. If there is not enough memory to create a new *neuron*, **MakeNeuron** signals an error and returns a null pointer. Neurons can be connected together using the

subroutine **ConnectNeuron** which takes pointers to two (not necessarily different) neurons as its arguments. This subroutine forms a synaptic connection *from* the first neuron, *to* the second neuron, thus creating a new axon for the first neuron, and a new dendrite for the second neuron. The appropriate synaptic lists for each neuron are accordingly updated. Listing A.4 gives the source code for **MakeNeuron** and **ConnectNeuron**.

New network flavors are specified by directly loading the list of neurons and required function pointers into an empty *net* template. No subroutine currently exists to assemble new networks as **MakeNeuron** does for neurons.

The network as a whole is manipulated by the subroutines **PulseNet(network)**, and by directly calling the teaching and debugging subroutines embedded in the network definition. **PulseNet(network)** updates the values of all neurons in the network by calling the subroutine **FireNeuron(neuron)**, which instructs each neuron to update itself using its activation function and its current net input, for each neuron. The code for both of these functions is listed in Listing A.5.

The dynamic simulation section of the program is a straightforward implementation of a fourth order, fixed stepsize Runge-Kutta algorithm, coded in the subroutine **RKInt(old_vector,new_vector,delta_t.)**. This subroutine takes as inputs the current state of the plant, **old_vector**, and the time step, **delta_t.**, and returns the results of integrating the plant dynamics over the specified interval in the vector **new_vector**. The plant dynamics are specified in the subroutine **UserFunction(in_vector, out_vector)**, where the relation:

$$\mathbf{out\_vector = f(in\_vector, control)}$$

holds between the two arguments, with f(•) the relation between the derivatives of the states (on the left hand side of the above equation), and the current states and control. Listing A.6 lists the C source code for these two functions; there are actually several sets of dynamics in the listed version of **UserFunction**, however all but one of these is commented out during any particular run of the program.

Finally, Listing A.7 shows the core of the NMC simulation. This program uses the structures and subroutines discussed above to implement the NMC algorithm as detailed in Chapter 2.0. The parameters which control the simulation, α, η, f, M_x, $M_{\dot{x}}$, x_d, M_u, n, and parameters specifying how much data should be recorded for each solo run and how many iterations of the algorithm should be performed, are contained in a formatted data file called NetPars and are read in at the start of the simulation by the subroutine **GetPars**. An example data file, and the source for **GetPars** is shown in Listing A.8.

Listing A.1: Object and Structure Templates for Network Simulation Facility

```
#define     MAXNET          99
#define     FANLIM          (MAXNET-1)

#define     INPUT           1
#define     HIDDEN          2
#define     OUTPUT          4
#define     BIAS            8


struct Network {
                struct Neuron *neurons[MAXNET];
                int output_vector[1][5];
                void (*Teacher)();
                void (*Displayer)();
                void (*Debugger)();
};

struct Neuron {
                float (*ActFunction)();
                float (*DerivActFunction)();
                float (*OutFunction)();
                int neuron_class;
                struct Synapse *dendrites;
                struct Synapse *axons;
                float net;
                float output;
                float error;
                float storage;
};

struct Synapse {
        float strength;
        float storage;
        struct Neuron *neighbor;
        struct Synapse *next_synapse;
};
```

Listing A.2: Back Propagation and Network Debugging Subroutines

```
#include <stdio.h>
#include "NetDefs.h"
#include <math.h>

void BackProp(net)
        struct Network net;
{
        extern int NETSIZE, PATTERNS;
        extern float LEARN_RATE, DECAY_RATE;
        register i = NETSIZE-2, j = 0;
        float new_error, temp_delta;
        struct Neuron *next_cell, *find_cell;
        struct Synapse *axon, *axon_tree, *dendritic_tree, *dendrite;
        static int passes = 0;
        extern float DerivActFunction();


/* First thing: make PATTERNS back passes through the net, adding up the
   total errors from each step.  When all the patterns have been presented
   and the total error from these computed for each neuron, divide by the
   total number of patterns to find the average amount of error for each
   pass, then adjust the connection strengths using the average error  */

/*          Process all the output neurons first!!      */

        while (i >= 0) {
                next_cell = net.neurons[i--];
                if (next_cell->neuron_class == OUTPUT) {
                        new_error = 0.0;
                        new_error += next_cell->storage;
/*                      new_error += (net.output_vector[passes][j++] - next_cell->output);
*/

                        new_error *= (*next_cell->DerivActFunction)(next_cell->output);
                        next_cell->error = new_error;
                }
        }
        i = NETSIZE-2;

/*  Compute the errors for all the other neurons    */

        while (i >= 0) {
                next_cell = net.neurons[i--];
                if (next_cell->neuron_class != OUTPUT) {
                        axon_tree = next_cell->axons;
                        if (axon_tree != NULL) {
                                axon = axon_tree;
                                new_error = 0;
                                do {
```

```
                                        new_error += (axon->neighbor->error)*(axon->strength);
                                    } while ((axon = axon->next_synapse) != NULL);
                                    new_error *= (*next_cell->DerivActFunction)(next_cell->output);

                                    next_cell->error = new_error;
                                }
                            }
                        }

/* Compute the weight changes required for each step and add them all up */

                        i = 0;
                        while ((next_cell = net.neurons[i]) != NULL) {
                            dendritic_tree = next_cell->dendrites;
                            if (dendritic_tree != NULL) {
                                dendrite = dendritic_tree;
                                do {
                                    temp_delta = (next_cell->error)*(dendrite->neighbor->output);

                                    temp_delta *= LEARN_RATE;
                                    temp_delta += DECAY_RATE*dendrite->storage;
                                    dendrite->strength += temp_delta;
                                    dendrite->storage = temp_delta;

/*  This unfortunate piece of code is necessary to keep the axon-dendrite
    information symmetric
                                                                        */

                                    j = 0;
                                    while ((find_cell = net.neurons[j++]) != NULL) {
                                        if (dendrite->neighbor == find_cell) {
                                            axon = find_cell->axons;
                                            do {
                                                if (axon->neighbor == next_cell) {
                                                    axon->strength = dendrite->strength;
                                                }
                                            } while ((axon = axon->next_synapse) != NULL);
                                        }
                                    }

/*  That's the end of that nonsense... */

                                } while ((dendrite = dendrite->next_synapse) != NULL);
                            }
                            i++;
                        }

                        if (++passes == PATTERNS) passes = 0;

                    }
```

Listing A.2 Continued...

```
#include <stdio.h>
#include "NetDefs.h"
#include <math.h>

void Dump(net)
        struct Network net;
{
        extern FILE *outfile;
        register i = 0, j = 0;
        struct Neuron *next_cell;
        struct Synapse *axon, *dendrite;

        while ((next_cell = net.neurons[i]) != NULL) {
                fprintf(outfile,"\n \n
**********************************************************");
                fprintf(outfile,"\n Stats for Neuron %d : \n", i);
                fprintf(outfile," Output: %f \t Error: %f \t Net Input: %f",
                        next_cell->output,next_cell->error, next_cell->net);

                axon = next_cell->axons;
                if (axon != NULL) {
                        j = 0;
                   do {
                           fprintf(outfile,"\n Axon     %d has strength: %f",j, axon->strength);

                           j++;
                        } while ((axon = axon->next_synapse) != NULL);
                }

                dendrite = next_cell->dendrites;
                if (dendrite != NULL) {
                        j = 0;
                        do {
                                fprintf(outfile,"\n Dendrite %d has strength: %f", j,
                                                dendrite->strength);
                                j++;
                         } while ((dendrite = dendrite->next_synapse) != NULL);
                }
                i++;
                fprintf(outfile,"\n
**********************************************************");
        }

}
```

Listing A.3: Activation Function Subroutines

```
#include <stdio.h>
#include <math.h>
float Sigmoidal(net)
        float net;
{
        return 1/(1+exp(-1.0*net));
}

float Identity(net)
        float net;
{
        return net;
}

float DerivSigmoidal(net)
        float net;
{
        return net*(1.0-net);
}

float AlwaysOn(net)
        float net;
{
        return 1.0;
}

float AlwaysOff(net)
        float net;
{
        return 0.0;
}

float Forced(net)
        float net;
{
        extern float input_vector[];
        extern int NUMINPUTS;
        static int i = 0;
        if (i == NUMINPUTS) i = 0;
        return input_vector[i++];
}

float Linear(net)
        float net;
{
        return net;
}

float DerivLinear(net)
        float net;
{
        return 1.0;
}
```

Listing A.4: Neuron Manipulation Subroutines

```
#include <stdio.h>
#include <malloc.h>
#include "NetDefs.h"


struct Neuron *MakeNeuron(NeuronClass,ActFunc,DActFunc,OutFunc)
int NeuronClass;
float (*ActFunc)(), (*DActFunc)(), (*OutFunc)();

{
        struct Neuron *temp_neuron;
        extern FILE *outfile;

        printf("Defining new neuron...");
        temp_neuron = (struct Neuron *) malloc(sizeof(struct Neuron));
        if (temp_neuron != NULL) {
                temp_neuron->ActFunction = ActFunc;
                temp_neuron->DerivActFunction = DActFunc;
                temp_neuron->OutFunction = OutFunc;
                temp_neuron->neuron_class = NeuronClass;

                temp_neuron->error = 0.0;
                temp_neuron->output = 0.5;
                temp_neuron->storage = 0.0;
                temp_neuron->net = 0.0;

                temp_neuron->dendrites = NULL;
                temp_neuron->axons = NULL;
                printf("done.\n");

                return temp_neuron;
        }
        else {
                printf("\n **No more memory to create neurons!!**\n");
                exit(1);
        }
}

void ConnectNeuron(neuron_1,neuron_2)
        struct Neuron *neuron_1,*neuron_2;
{
        struct Synapse *new_axon, *new_dendrite, *axon, *dendrite;
        extern float nrand();
        extern FILE *outfile;

        printf("Connecting neurons...");
        new_axon = (struct Synapse *) malloc(sizeof(struct Synapse));
        new_dendrite = (struct Synapse *) malloc(sizeof(struct Synapse));
        if ((new_axon != NULL) && (new_dendrite != NULL)) {
```

```
                new_axon->storage = new_dendrite->storage = 0.0;
                new_axon->next_synapse = new_dendrite->next_synapse = NULL;
                new_axon->strength = new_dendrite->strength = nrand(.3);
                new_axon->neighbor = neuron_2;
                new_dendrite->neighbor = neuron_1;

/* If the "from" neuron doesn't have any axons yet (axon list is NULL),
        then make this synapse the head of the axon list                */

                if (neuron_1->axons == NULL) {
                        neuron_1->axons = new_axon;
                }

/* ...otherwise, look for the end of the "from" neurons axon list and
        add the new synapse.        */

                else {
                        int i = 0;

                        axon = neuron_1->axons;
                        while (axon->next_synapse != NULL) {
                                axon = axon->next_synapse;
                        }
                        axon->next_synapse = new_axon;
                }

/*  Now do the same thing for the "to" neuron's dendritic tree        */

                if (neuron_2->dendrites == NULL) {
                        neuron_2->dendrites = new_dendrite;
                }
                else {
                        int i = 0;

                        dendrite = neuron_2->dendrites;
                        while (dendrite->next_synapse != NULL) {
                                dendrite = dendrite->next_synapse;
                        }
                        dendrite->next_synapse = new_dendrite;
                }
                printf("done.\n");

        }

        /* Abort if out of memory   */
        else {
                printf("\n **No memory availible to create a new synapse!!**\n");
                exit(1);
        }
}
```

Listing A.5: Network Manipulation Subroutines

```
#include "NetDefs.h"
#include <stdio.h>


void PulseNet (net_type)
        struct Network net_type;
{
        register i = -1;
        struct Neuron *next_cell;
        extern void FireNeuron();

        while ((next_cell = net_type.neurons[++i]) != NULL) {
                FireNeuron(next_cell);
                next_cell->output = next_cell->storage;
        }

}

void FireNeuron(neuron_type)
        struct Neuron *neuron_type;
{
        struct Synapse *synaptic_tree, *synapse;
        register i = -1;
        float sum = 0.0, activation = 0.0;

        synaptic_tree = neuron_type->dendrites;
        if (synaptic_tree != NULL) {
                synapse = synaptic_tree;
                do {
                        sum += synapse->strength*(synapse->neighbor->output);
                } while ((synapse = synapse->next_synapse) != NULL);
        }
        neuron_type->net = sum;
        activation = (*neuron_type->ActFunction)(sum);
        neuron_type->storage = (*neuron_type->OutFunction)(activation);
}
```

Listing A.6: Dynamic System Simulator Subroutines

```
#include "NetDefs.h"
#include <stdio.h>
#include <math.h>
#define MAXDIM 20

void RKInt(old_vector,new_vector,delta_t)
        float old_vector[], new_vector[], delta_t;
{
        extern int DIMENSION;
        float k0[MAXDIM], k1[MAXDIM], k2[MAXDIM], k3[MAXDIM];
        float temp_vector[MAXDIM], temp;
        register i = 0;

        extern void UserFunction();

/*-------------------------Begin Main Code-----------------------------*/

        UserFunction(old_vector,k0);
        for (i = 0; i <= DIMENSION-1; i++)
                temp_vector[i] = old_vector[i]+0.5*delta_t*k0[i];

        UserFunction(temp_vector,k1);
        for (i = 0; i <= DIMENSION-1; i++)
                temp_vector[i] = old_vector[i]+0.5*delta_t*k1[i];

        UserFunction(temp_vector, k2);
        for (i = 0; i <= DIMENSION-1; i++)
                temp_vector[i] = old_vector[i]+delta_t*k2[i];

        UserFunction(temp_vector,k3);
        for (i = 0; i <= DIMENSION-1; i++) {
                temp = 0.166666667*delta_t*(k0[i]+2*k1[i]+2*k2[i]+k3[i]);
                new_vector[i] = old_vector[i]+temp;
        }
}

void UserFunction(in_vector, out_vector)
        float in_vector[], out_vector[];
{
        extern int NOW, sgn();
        extern float StairCase();
        static float c1 = 0.0, c2 = 0.0, c3 = 1.0, foo = 0.0;
        extern float control;
        extern FILE *outfile;

/*              Bang-Bang actuation filter              */

/*      if (control>1.0) foo = 20.0;
                else if (control< -1.0) foo = -20.0;
                else foo = 0.0;   */
```

```
/*              Arbitrary second order linear dynamics          */

        out_vector[0] = in_vector[1];
        out_vector[1] = c1*in_vector[0]+c2*in_vector[1]+c3*control;


/*                      MPOD dynamics                           */

/*      out_vector[1] = -2.5*in_vector[1]*fabs(in_vector[1])
                        - 0.4*sin(in_vector[0]) + c3*StairCase(control); */
}

float StairCase(cons)
        float cons;
{
        static float mag = 18.0;

        if (cons > 16.0) return mag*16.0;
          else if (cons < -16.0) return -mag*16.0;
          else return ((int) cons) * mag;
}

int sgn(x)
        float x;
{
        if (x < 0) return -1;
          else if (x > 0) return 1;
          else return 0;
}
```

Listing A.5: Main Simulator Program

```
#include <stdio.h>
#include "NetDefs.h"
#include <math.h>

float input_vector[5];
float control = 0.0;
int NOW = 0;
FILE *outfile, *outfile2;

int NETSIZE, PATTERNS, NUMINPUTS, NUMOUTPUTS, DIMENSION, MAXPASSES;
int SAMPLE_RATE, OUTPUT_RATE;

float LEARN_RATE, DECAY_RATE, TMAX, XD[5], M[5], MU, MAXCON, NEXP;
float DELTA_T;

main () {

        struct Neuron *neuron[25];
        struct Network net_a;

        extern float Sigmoidal(), Identity(), AlwaysOn(), AlwaysOff();
        extern float Forced(), DerivSigmoidal(), Linear(), DerivLinear();
        extern float DesiredResponse(), nrand(), AlwaysOn5(), StairCase();
        extern void BackProp(), Dump(), RKInt();
        extern struct Neuron *MakeNeuron();
        extern void ConnectNeuron();
        extern float input_vector[];
        extern int NOW;
        int i,j,k,l, bomb=0;

        float pattern[1][5], response = 0.0, foo = 0.0, goo = 0.0;
        float state[2], new_state[2], delta_t,time,error,ic[2];
        float con_stack[20];
        extern float control;
        extern FILE *outfile, *outfile2;

        extern int MAXPASSES, PATTERNS, NETSIZE, SAMPLE_RATE, OUTPUT_RATE;
        extern int NUMINPUTS, DIMENSION;
        extern float DELTA_T, TMAX, NEXP;


/* Start the simulation */

        outfile = fopen("DebugOutput", "w");
        outfile2 = fopen("NetOutput","w");
        printf("Beginning simulation: \n \n");
        getpars();

/* Specify the network interconnections */
```

```
/*              Create the individual neurons      */

        neuron[0] = MakeNeuron(INPUT, Forced, DerivLinear, Identity);
        neuron[1] = MakeNeuron(INPUT, Forced, DerivLinear, Identity);
        neuron[2] = MakeNeuron(HIDDEN, Sigmoidal, DerivSigmoidal, Identity);
        neuron[3] = MakeNeuron(HIDDEN, Sigmoidal, DerivSigmoidal, Identity);
        neuron[4] = MakeNeuron(HIDDEN, Sigmoidal, DerivSigmoidal, Identity);
        neuron[5] = MakeNeuron(BIAS, AlwaysOn, DerivSigmoidal, Identity);
        neuron[6] = MakeNeuron(BIAS, AlwaysOn, DerivSigmoidal, Identity);
        neuron[7] = MakeNeuron(BIAS, AlwaysOn, DerivSigmoidal, Identity);
        neuron[8] = MakeNeuron(OUTPUT, Linear, DerivLinear, Identity);
        neuron[9] = MakeNeuron(BIAS, AlwaysOn, DerivSigmoidal, Identity);
/*      neuron[10] = MakeNeuron(INPUT, Forced, DerivLinear, Identity);  */

/*              Stick the neurons into the network definition    */

        for (i = 0; i <= NETSIZE-2; i++) net_a.neurons[i] = neuron[i];
        net_a.neurons[NETSIZE-1] = NULL;

/*                      Hook the neurons together               */

        for (i = 0; i <= 1; i++) {
                for (j=2; j<=4; j++) ConnectNeuron(net_a.neurons[i], net_a.neurons[j]);
                ConnectNeuron(net_a.neurons[i],net_a.neurons[8]);
        }
        for (i = 2; i<= 4; i++) ConnectNeuron(net_a.neurons[i],net_a.neurons[8]);
        for (i = 5; i<=7; i++) ConnectNeuron(net_a.neurons[i],net_a.neurons[i-3]);
        ConnectNeuron(net_a.neurons[9],net_a.neurons[8]);

/*      for (i=2; i<=4; i++)
                ConnectNeuron(net_a.neurons[10], net_a.neurons[i]);
        ConnectNeuron(net_a.neurons[10], net_a.neurons[8]);  */

/*              Establish the rest of the network parameters            */

        net_a.Teacher = BackProp;
        net_a.Debugger = Dump;

/*              Show the state of the network before training                   */

        (*net_a.Debugger)(net_a);
```

```
/*----------------New Adaptive Control algorithm starts here----------*/

        delta_t = DELTA_T;

/*              Loop through all the requested iterations       */

        for (j = 1; j <= MAXPASSES; j++) {

/*              Zero out state and set plant to initial conditions          */

                ic[0] = 0.0;
                ic[1] = 0.0;
                for (k = 1; k <= DIMENSION; k++) state[k-1] = ic[k-1];
                error = 1.0;
                response = 0.0;
                time = 0.0;

/*              Loop until the requested time passes   */

                for (i = 0; time <= TMAX; i++) {

                        time = i*delta_t;

/*      Present input vector to the net and determine the net outputs  */

                        input_vector[0] = state[0];
                        input_vector[1] = state[1];
                        PulseNet(net_a);

/*      Uncomment this next section to put a delay in the control loop    */

/*                      control = con_stack[0];
                        for (k = 0; k <= 18; k++) con_stack[k] = con_stack[k+1];
                        con_stack[19] = net_a.neurons[8]->output;   */

                        control = net_a.neurons[8]->output;

/*              Integrate the plant forward in time       */

                        RKInt(state, new_state, delta_t);
                        for (k = 1; k <= DIMENSION; k++) state[k-1] = new_state[k-1];
                        response = state[0];
```

```
/*              Compute the payoff function             */

                        foo = 0.0;
                        for (k = 0; k <= DIMENSION-1; k++) {
/*                              if (XD[k] == 0.0) goo = 1.0;
                                  else goo = XD[k];    */
                                goo = 1.0;
                                foo += M[k]*((XD[k]-state[k])/goo);
                        }
                        foo += -sgn(control)*MU*pow( (double) (control/MAXCON),
                                                (double) NEXP);
                        net_a.neurons[8]->storage = foo;

/*              Check if any bounds are exceeded                */

                        if ( (fabs(state[0]) >= 10.0)
                                || (fabs(control) > 5*MAXCON) )
                                {


/*      YES!! So shut down the plant and network and restart the simulation.
                 If catastrophic, stop the program  */

                                printf("\n\n ERROR!!...\n");
                                if (error==0.0) {
                                        printf("\n \n Error has occurred...\n");
/*                                      net_a.neurons[8]->storage = -1.0*sgn(state[0])*50.0;
                                        (*net_a.Teacher)(net_a);  */
                                        exit(1);
                                }
                                for (k=1;k<=DIMENSION;k++) state[k-1] = ic[k-1];
                                response = 0.0;
                                error = 0.0;
                        }
                        else {

/*      NO.  So check if this is an update step, and if so call the teacher  */

                                if ((i%SAMPLE_RATE) == 0) {
                                  (*net_a.Teacher)(net_a);
                                  error++;
                                }
                        }
                }

                printf("%3d...",j);
/*              if (j != MAXPASSES) continue;  */
```

Listing A.7 Continued...

```
/*                      Now set up to run solo                          */

                fprintf(outfile,"\n \n %d Running without teacher...\n",j);

                for (k = 1; k <= DIMENSION; k++) state[k-1] = ic[k-1];
                response = 0.0;
                time = 0.0;
                fprintf(outfile,"Time\tControl\t   Output\t   Velocity\n");

                for (i = 0; time <= TMAX; i++) {
                        time = i*delta_t;
                        input_vector[0] = state[0];
                        input_vector[1] = state[1];
                        PulseNet(net_a);

                        control = net_a.neurons[8]->output;

/*              Uncomment this to put a delay in the control loop       */

/*                      control = con_stack[0];
                        for (k = 0; k <= 18; k++) con_stack[k] = con_stack[k+1];
                        con_stack[19] = net_a.neurons[8]->output; */


/*                Print out as often as requested                       */

                        if ((i%OUTPUT_RATE) == 0) {
                          fprintf(outfile, "%f\t%f\t%f\t%f\n",
                                  time, control, response, state[1]);
                        }


/*                Integrate the plant forward in time                   */

                        RKInt(state, new_state, delta_t);
                        for (k = 1; k <= DIMENSION; k++) state[k-1] = new_state[k-1];
                        response = state[0];

                }
        }

/*              Show the network at the end of all the iterations       */

        (*net_a.Debugger)(net_a);

}
```

Listing A.8: Simulation Parameter Subroutines

```
#include <stdio.h>

FILE *infile;

void getpars() {
        extern int NETSIZE, PATTERNS, NUMINPUTS, NUMOUTPUTS,
DIMENSION;
        extern int MAXPASSES, SAMPLE_RATE, OUTPUT_RATE;

        extern float LEARN_RATE, DECAY_RATE, XD[], M[], MAXCON, MU;
        extern float DELTA_T, TMAX, NEXP;

        extern FILE *infile;
        extern void gotopar();

        register i;

/*              Load in the simulation parameters     */

        infile = fopen("Netpars","r");

        gotopar();
        fscanf(infile,"%d \n", &NETSIZE);
        gotopar();
        fscanf(infile,"%d \n", &PATTERNS);

        gotopar();
        fscanf(infile, "%d \n", &NUMINPUTS);
        gotopar();
        fscanf(infile, "%d \n", &NUMOUTPUTS);
        gotopar();
        fscanf(infile, "%d \n", &DIMENSION);

        gotopar();
        fscanf(infile, "%f \n", &TMAX);
        gotopar();
        fscanf(infile, "%f \n", &DELTA_T);
        gotopar();
        fscanf(infile, "%d \n", &MAXPASSES);
        gotopar();
        fscanf(infile, "%d \n", &OUTPUT_RATE);

        gotopar();
        fscanf(infile, "%f \n", &LEARN_RATE);
        gotopar();
        fscanf(infile, "%f \n", &DECAY_RATE);
        gotopar();
        fscanf(infile, "%d \n", &SAMPLE_RATE);
```

```
        for (i = 0; i <= DIMENSION-1; i++) {
                gotopar();
                fscanf(infile, "%f \n", &XD[i]);
        }
        for (i = 0; i <= DIMENSION-1; i++) {
                gotopar();
                fscanf(infile, "%f \n", &M[i]);
        }
        gotopar();
        fscanf(infile, "%f \n", &MAXCON);
        gotopar();
        fscanf(infile, "%f \n", &MU);
        gotopar();
        fscanf(infile, "%f \n", &NEXP);

        printf("Simulation Parameters: \n \n");
        printf("NETSIZE = %d, PATTERNS = %d \n", NETSIZE, PATTERNS);
        printf("NUMINPUTS = %d, NUMOUTPUTS = %d \n", NUMINPUTS,
NUMOUTPUTS);
        printf("DIMENSION = %d, MAXPASSES = %d \n", DIMENSION,
MAXPASSES);
        printf("TMAX = %f, DELTA_T = %f \n", TMAX, DELTA_T);
        for (i = 0; i <= DIMENSION-1; i++) printf("XD[%d] = %f\t", i, XD[i]);
        printf("\n");
        for (i = 0; i <= DIMENSION-1; i++) printf("M[%d] = %f\t", i, M[i]);
        printf("\n");
        printf("MAXCON = %f  MU = %f   NEXP = %f \n", MAXCON, MU, NEXP);
        printf("LEARN_RATE = %f, DECAY_RATE = %f \n", LEARN_RATE,
DECAY_RATE);
        printf("SAMPLE_RATE = %d, OUTPUT_RATE = %d \n", SAMPLE_RATE,
OUTPUT_RATE);

}

void gotopar() {

        extern FILE *infile;
        char c;

        while ((c = fgetc(infile)) != ':') {
                if (feof(infile)) {
                  printf("Input file abnormally terminated.  Exiting. \n");
                  exit(1);
                }
        }
        fgetc(infile);
}
```

Listing A.7 Continued...

VARIABLE	DEFINITION	VALUE
(NETSIZE)	Number of neurons in the network	: 11
(PATTERNS)	Total patterns for the net to learn	: 1
(NUMINPUTS)	Number of signals input to the net	: 2
(NUMOUTPUTS)	Number of signals output by the net	: 1
(DIMENSION)	Dimension of the simulated plant	: 2
(TMAX)	Length of each simulated run	: 20.0
(DELTA_T)	Time step size for integration	: 0.005
(MAXPASSES)	Number of simulation runs	: 50
(OUTPUT_RATE)	How often data should be recorded	: 10
(LEARN_RATE)	Rate at which connections are changed:	: 0.25
(DECAY_RATE)	Momentum term for weight changes	: 0.50
(SAMPLE_RATE)	How often to teach the net	: 10
(XD)	Desired final state vector	: 1.00 : 0.00
(M)	Weightings for state deviations	: 1.00 : 0.50
(MAXCON)	Maximum allowable control force	: 16.0
(MU)	Weighting for excessive control usage:	: 0.3
(NEXP)	Exponent for normalizing control	: 4