Summer 1996

Neural Generalized Predictive Control for Real-Time Control

Donald I. Soloway
Old Dominion University

NEURAL GENERALIZED PREDICTIVE CONTROL FOR REAL-TIME CONTROL

Donald I. Soloway

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

ELECTRICAL ENGINEERING

OLD DOMINION UNIVERSITY
August 1996

Approved by:

Oscar R. González (Director)

Stephen Zahorian

Thomas E. Alberts

ABSTRACT

In this thesis a computationally efficient Generalized Predictive Control (GPC) algorithm is presented and implemented. The algorithm is more efficient than others because the number of iterations needed for convergence is significantly lower with Newton-Raphson. The main additional cost with Newton-Raphson algorithm is the calculation of the Hessian. This overhead is not a problem because of the reduced number of iterations, making the algorithm suitable for real-time control. For nonlinear control applications, a neural network is used as a dynamical system predictor leading to a Neural Generalized Predictive Control (NGPC) algorithm which is presented in detail in this thesis. An advantage of GPC is that physical constraints can be easily incorporated. This thesis includes hard actuator constraints using a differentiable function. A real-time procedure to control unstable plants with an untrained neural network is also presented. In this case the neural network is initialized with an embedded linear model of the process.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Dr. Oscar R. González, for his help and support, to Dr. Thomas Alberts and Dr. Stephen Zahorian for serving on my thesis committee, and a special thanks to my co-investigator and friend, Pam Haley, for the help she has given me over the years in the development, implementation and testing of the Neural Generalized Predictive Controller. I would also like to thank my parents Sidney and Rhoda Soloway for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

Artificial neural networks are algorithms or devices that attempt to behave like very simple organic brains. Their ability to generalize their learned patterns has shown to have broad applications. These applications range from character recognition for postal codes to control of robotic manipulators for space applications. The use of neural networks since their inception in the 1940's has been limited until the last decade because of inherent limitations of computational technologies, and shortcomings in the theory of neural networks.

The 1980's saw a renewed interest in artificial neural networks with the works of Hopfield [15], Kohonen [16], and Rumelhart [5] as well as others. Hopfield showed how to design a neural network to quickly solve a constraint satisfaction problem. His network is able to solve optimization problems like the traveling salesman problem. Kohonen developed a self organizing network for pattern clustering. Rumelhart, et al., independently redeveloped[1] a learning rule for a multi-layer network. Their work led researchers to apply neural networks to various problems, such as Sejnowski's [6] work

[1] This learning rule has been independently developed in several disciplines in the last 30 years [42]. For example, it appears in Paul Werbos' 1974 doctoral dissertation. When the PDP books audience grew Werbos work resurfaced.

on converting English text to speech, Anderson's [1] work on control of an inverted pendulum, and Grossberg and Kuperstein's work on sensory-motor control [7].

The use of neural networks for nonlinear control applications has been of great interest in the last several years [2,3]. The phenomenal growth of neural control algorithms is due to the ability of neural networks to model nonlinear systems [4]. These neural models or system estimators can then be used with model-based control algorithms. The early controllers were based on inverse control techniques [38]. Inverse controllers, in general, suffer from sensitivity to plant model and disturbances and are limited to the control of stable minimum-phase plants. To overcome the sensitivity problem with inverse controls, neural networks were merged with Internal Model Control [24]. To handle non-minimum phase plants, unstable plants, and to handle the sensitivity problems, neural network based Generalized Predictive Control was developed. Several approaches to neural network based Generalized Predictive controllers have appeared in the literature since the early 1990's. The main difference between these approaches is the numerical technique used to minimize a quadratic cost function with or without constraints. The reported techniques in the literature are computationally costly which restricts their application to slower processes. The results in this thesis lead to real-time implementation of neural network based Generalized Predictive controllers for a larger class of processes.

The main objective and contribution of this thesis is a computationally efficient Neural Generalized Predictive Controller (NGPC) utilizing Newton-Raphson optimization algorithm to minimize the GPC cost function which was augmented to handle actuator saturation. In addition, a real-time technique is to control unstable starting with an untrained neural network. In this case the neural network is initialized with an embedded linear model of the plant. The real-time capabilities of the algorithm are evaluated with a timing analysis of the algorithm. Two case studies are presented to demonstrate the speed and control capability of NGPC.

To make the neural network based Generalized Predictive Control with input constraints computationally efficient, the Newton-Raphson iterative algorithm is used to perform the cost function minimization. The Newton-Raphson algorithm uses both first and second derivatives of a cost function to calculate the iterative step size. Furthermore, since the Newton-Raphson does not require any step size gain, no tuning of the minimization algorithm is needed. Additionally, the Newton-Raphson is the fastest general purpose converging nonlinear optimization algorithm [13,14]. A differentiable function is proposed to model the hard input constraint. This function can approximate the hard constraint to any desired degree of accuracy.

In general, dynamical plants can have multiple inputs and outputs (MIMO). This thesis will consider only a single input and a single output system (SISO). All references

to dynamical systems will be in the context of a discrete-time SISO plant. The techniques presented, however, can be generalized to MIMO plants.

The outline of this thesis is a follows: Chapter 2 develops the fundamental background, which includes: a description of a multi-layer feedforward neural network, neural network modeling of a dynamical system, and an overview of neural network control systems. In Chapter 3, an introduction to generalized and neural generalized predictive controls, a review of previous work on neural generalized predictive control, and the development of the cost function minimization procedure is presented. Chapter 4 demonstrates the use of this approach by applying the NGPC to several problems. Section 5 is the conclusion and a discussion of future work.

2. Background

2.1. Introduction

Neural networks are a collection of processing elements called nodes. Each node performs a local operation on the data presented to it. Typically each node performs the same operation, just on different data or connections. The connections can be from the external environment or from other nodes, including itself. This work started in the 1940's with McCulloch and Pitts studying the capabilities of the interconnections of several basic neuron models. Rosenblatt in 1958 developed a network architecture based on a neuron model that he called Perceptron. This work received much attention. Minsky and Papert in 1969 published a book, Perceptrons [39], where they proved several properties, and also pointed out several limitations. These limitations hampered funding levels until 1986, when the parallel distributed processing group published a set of algorithms for more complex networks then the Perceptron. These algorithms overcame the limitations pointed out by Minsky and Papert. Since then there has been an exponential growth in the interest and use of neural networks.

2.2. Organization of Chapter

This chapter introduces neural networks and their use in control systems. Section 2.3 develops the multi-layer feedforward network. Section 2.4 discusses the use of a multi-layer feedforward network as a universal function approximator and how to train this network. This section also describes one of the architectures for modeling dynamical systems. Section 2.5 introduces the use of neural networks for control systems.

2.3. Introduction to Neural Networks

2.3.1. Introduction

Neural networks perform a nonlinear mapping, for example, from an n-dimensional space to an m-dimensional space. These spaces may be spatial and/or temporal, such as the Euclidean space $\Re^n$ and the space of discrete-time signals l_2^n. In particular, neural networks has the capability of modeling dynamical systems. There are many ways one can define a neural network, but all neural networks have eight major characteristics in common [5]. They are comprised of

- a set of processing nodes,
- a state of activation for each node,
- an output function for each node,
- a pattern of connectivity among the nodes,
- a propagation rule for propagation of activities through the network,

- an activation rule for each node,
- a learning rule whereby patterns of connectivity are modified by experience, and
- an environment within which the network must operate.

Depending on the choice of these characteristics, different neural network paradigms arise. Hence, a discussion of each characteristic will be presented using a multi-layered feedforward neural network with the error Backpropagation paradigm.

2.3.2. The Node

A node is a processing element that maps an N^0-dim input vector to a scalar. A schematic of the node activity is given in Figure 1.

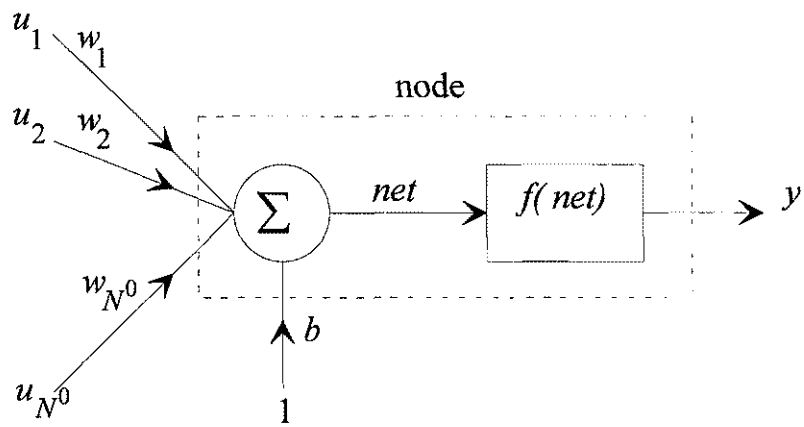

Figure 1. Schematic of a Node

In the first stage the N^0-dim vector of inputs is reduced to a scalar by performing an inner product between the input vector $\left[u_1 \;\; u_2 \cdots u_{N^0}\right]^T$ and the weight vector $\left[w_1 \;\; w_2 \cdots w_{N^0}\right]$ and adding a bias b. This operation is represented by

$$net = \sum_{i=1}^{N^0} w_i u_i + b \tag{1}$$

which gives the state of activation of the node. In the second stage, the output of the node is given by

$$y = f(net), \tag{2}$$

where f is the output function. There are several choices for the output function. The two most typically functions used are Gaussian and Sigmodal. The Gaussian is used for localized learning where the Sigmodal is used for global learning [30]. The Sigmodal function will be used here for global learning. The form of the Sigmodal function will be the hyperbolic tangent function.

2.3.3. The Layer

A layer consists of a set of nodes. This collection of nodes maps the N^i-dim vector of inputs to the N^{i+1}-dim vector of outputs. A schematic of the layer activity is given in Figure 2.

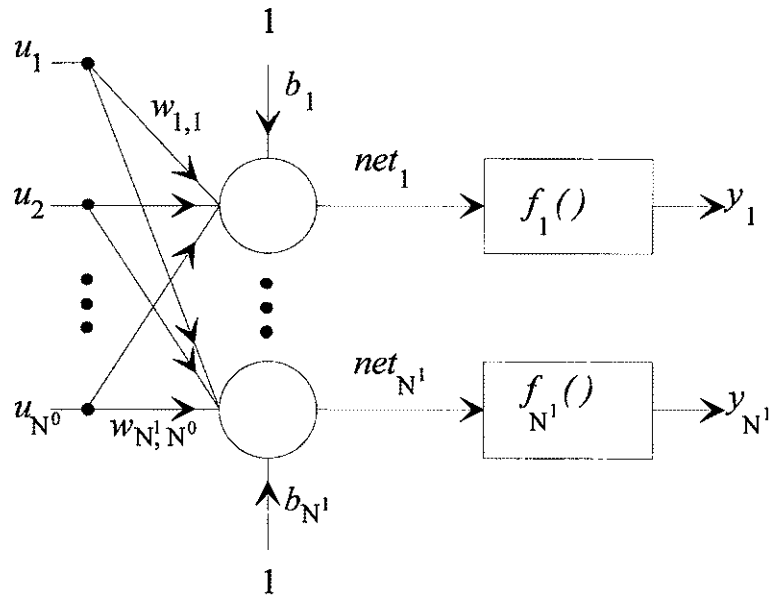

Figure 2. Layer of a Neural Network.

In the first stage, the node activities are calculated using (1) and (2). This activity can be simply represented by the product of the input vector $\left[u_1 \;\; u_2 \cdots u_{N^0}\right]^T$ and a weight matrix **W**, the addition of a bias vector **B**, and the transformation by **F**. The equations that describe this activity are

$$net_j = \sum_{i=1}^{N^0} w_{j,i} u_i + b_j \tag{3}$$

and

$$y_j = f_j\left(net_j\right) \quad \text{for } j=1,2,...N^1 \tag{4}$$

where

u_i is the i^{th} element of the input vector of length N^0,

$w_{j,i}$ is the weight connecting the i^{th} input, u_i, with the j^{th} output, y_j,

b_j is a bias input to the j^{th} node,

f_j is the output function of the j^{th} node, and

y_j is the output of the j^{th} node.

2.3.4. The Network

A multi-layer feedforward neural network is broken down into three parts: the input layer, the hidden layers, and the output layer. The input layer distributes the inputs to the following layer. The input layer does not multiply any weights nor does it process the inputs through an output function. The hidden and output layers consist of processing nodes. Each layer is fully connected via connection weights to the next layer as shown in Figure 3. This network has L layers with N^l nodes on layer l. The input layer is denoted as layer 0.

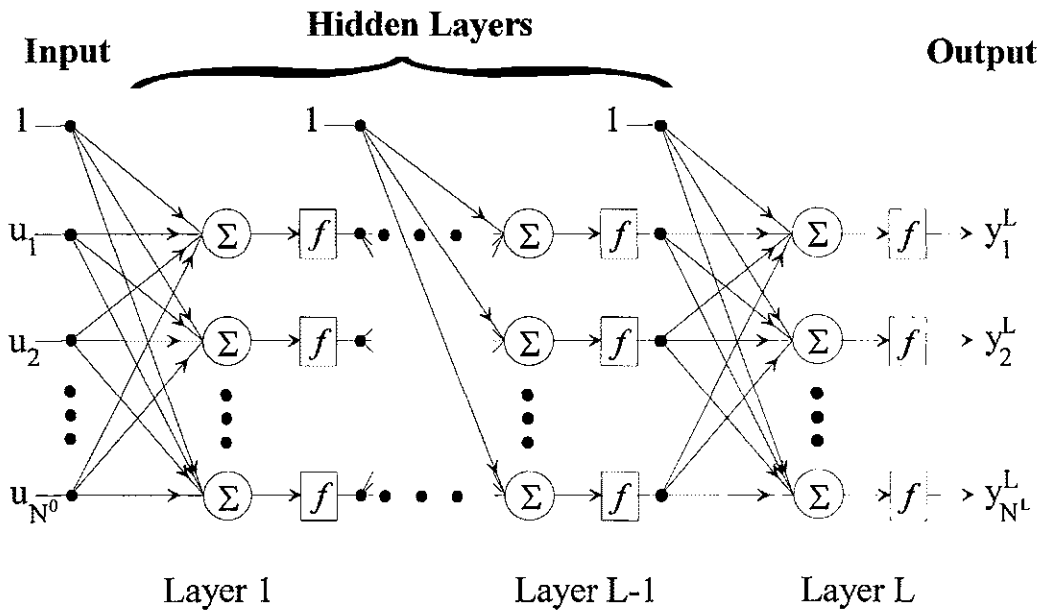

Figure 3. Multi-layer Feed Forward Neural Network

Forward propagation is accomplished by presenting an input, $\left[u_1 \;\; u_2 \cdots u_{N^0}\right]^T$, to the network. This input is fanned out to the first hidden layer and propagated through the nodes by evaluation of

$$net_j^l = \sum_{i=1}^{N^{l-1}} w_{j,i}^l y_i^{l-1} + b_j^l \text{ and} \tag{5}$$

$$y_j^l = f_j^l\left(net_j^l\right), \quad \text{for } j=1,2,...N^l \text{ and } l=0,1,...L, \tag{6}$$

where

L is the number of layers,

N^l is the number of nodes on the l^{th} layer, l=0,1,2,...,L,

y_j^l is the output of the j^{th} node of the l^{th} layer (if l=0 then $y_j^0=u_j$),

u_j is the j^{th} element of the input node with a vector of length N^0,

$w_{j,i}^l$ is the j,i^{th} element of the weight of the l^{th} layer with a matrix of size $N^{l+1}xN^l$,

b_j^l is a bias input to the j^{th} node of the l^{th} layer, and

f_j^l is the output function of the j^{th} node of the l^{th} layer.

This result is fanned out to the next layer, and the process is repeated until the output layer L is reached (see Figure 3).

2.4. Neural Network Input/Output Estimators

2.4.1. Introduction

In Section 2.3 the multi-layer feedforward network architecture was described. The main property of this network is its ability to model input/output (I/O) relationships. This network has been shown to be a universal function approximator. Adding a time series structure to the static network converts the network to a dynamical system estimator. These features are described in the following subsections. Section 2.4.2 describes this property for neural network function approximators. Section 2.4.3 describes the training rule for the network as a function approximator. Section 2.4.4 describes neural networks as dynamical system estimators.

2.4.2. Neural Networks as Function Approximators

The function approximation problem can be posed as follows. Given a set of I/O data, $\left[u_1 \quad u_2 \cdots u_{N^0}\right]^T, \left[y_1 \quad y_2 \cdots y_{N^L}\right]^T$ and a multi-layer feedforward network, find a set of weights and biases that approximate the I/O relationship. The first question that arises is how good could this approximation be. Hornik and White showed that a multi-layer feedforward network with one hidden layer can approximate to any desired degree of accuracy measurable and continuous functions. The second question is then how does one obtain the weights and biases for this approximation. This is accomplished by the minimization of a cost function. The process of minimizing the cost function is called

training a network. The algorithm is called the training or learning rule. There are several learning rules that have been developed since 1986. The one presented in the next section is based on the gradient descent technique.

2.4.3. Backpropagation Training of The Network

Minsky and Papert pointed out limitations of Rosenblatt's Perceptrons. One of these limitations was that there was no learning rule for a multilayer Perceptron architecture. Then in 1986 the PDP Group published an algorithm for training a multi-layer feedforward network [5]. Several different rules to update the weight and biases have been developed for feedforward networks ([17], [18] and [19]). The one used here is based on a gradient descent algorithm called Backpropagation [5]. The Backpropagation algorithm minimizes the root mean squared (RMS) error (7) with respect to the weights and biases of the network. The RMS error is a function of the difference between the desired output and the network output over all the training pairs. It is given by

$$RMS = \frac{1}{PN^L}\sqrt{\sum_{p=1}^{P}\sum_{j=1}^{N^L}\left(y_{jp} - y_{jp}^L\right)^2}, \tag{7}$$

where

P is the number of input/output training pairs,

L is the number of layers,

N^L is the number of nodes on the output layer L,

y_{jp} is the desired output for the j^{th} output node of the p^{th} training pattern, and

y_{jp}^{L} is the output of the j^{th} node of the L^{th} layer for the p^{th} training pattern.

When the minimization of the RMS error is performed an update rule is obtained for the weights and biases. This update rule is applied to each input/output training pair. The pattern notation p is dropped for simplicity. Thus the update equation is

$$\Delta w_{ji}^{l} = \eta \delta_{j}^{l} y_{i}^{l-1} \tag{8}$$

and

$$\delta_{j}^{l} = \begin{cases} \dot{f}_{j}^{l}\left(net_{j}^{l}\right)\sum_{j=1}^{N^{l+1}} \delta_{j}^{l+1} w_{ji}^{l+1} & , \quad \text{for } l \neq L \\ \dot{f}_{j}^{L}\left(net_{j}^{L}\right)\left(y_{j} - y_{j}^{L}\right) & , \quad \text{for } l = L \end{cases} \tag{9}$$

where

L is the number of layers,

N^{l+1} is the number of nodes on the layer l+1,

η is the learning gain,

$\Delta w_{j,i}^{l}$ is the change in the i^{th},j^{th} weight of the l^{th} layer,

y_{j}^{l} is the output of the j^{th} node of layer l (if l=0 then y_{j}^{0}=u_j),

u_j is the j^{th} element of the input node with a vector of length N^0,

$w_{j,i}^{l}$ is the j,i^{th} element of the weight of layer l with a matrix of size N^{l+1}xN^{l}, and

$\dot{f}_{j}^{l}(*)$ is the derivative of the output function for the j^{th} node of layer l with respect to net_{j}^{l}.

The biases of the network can be viewed as a weight with an input of one. Thus the update rule for the biases is the same as for the weights with the input to the node equal to one. In the rest of this thesis all references and comments made about the weights will also apply to the biases.

The weights and biases are initialized with small random numbers to avoid a problem called "breaking symmetry". The symmetry problem occurs when some or all of the weights and biases are the same value. The update rule will then update all of the same weights with the same value. This will limit the learning ability of the network [5]. The rule used here is to initialize these values with uniformly distributed random numbers between -0.01 and 0.01 divided by the number of weights connecting to the node. This normalization will avoid saturation of a node when the first few inputs are presented. If a node saturates, the derivative at that point is close to zero, and thus very small updating of the weights occurs. This initialization has proven successful when training a network as a dynamical system.

The Backpropagation algorithm is a special case of a gradient descent algorithm. As such, it is an iterative nonlinear first-order unconstrained optimization technique. The performance of the algorithm depends on the initial weight values, the learning rule parameter, and the technique used to update the weights. There are two techniques to update the weights with the Δw's. The first technique, called batch learning, obtains the

Δw's for all of the input/output training pairs, averages them together, and then changes all the weights of the network. The other technique, called on-line learning, updates the weights after obtaining the Δw for a single input/output pair. The latter technique is more applicable to control systems work because one typically does not have a predefined training set, but a set is generated while controlling a plant.

The scale factor, η, is known as the learning gain. This gain adjusts the rate of convergence of the network to the desired output. If the gain is small, the weight updates are small and convergence to a good solution is slow. If the gain is large, the weight's updates are large and this could lead to oscillations. Improving the performance of the training algorithm is still an ongoing area of research. Techniques have been proposed to initialize the weights and to adaptively update the learning rate.

2.4.4. Neural Networks as Dynamical Plant Estimators

2.4.4.1. Introduction

In Sections 2.4, we showed the network architecture and the learning rule for a neural network to be used as a universal approximator for static mappings. The same universal approximation property can lead to dynamical plant estimators if the appropriate dynamical structure is added to the neural network. Since we are interested in neural networks implemented in a computer, discrete-time dynamical plant models will be obtained.

For a network to model a dynamical plant, a structure must be added to the network to capture the effects of previous inputs and/or outputs or state information. Several techniques, based on linear digital filter principles have proven particularly useful. One technique assigns the input of the network to the states of the plant. This works when all the states of the plant are measured. Another technique adds recursion to the network by taking the output of a node and connecting it back to itself. Different variations of this idea can be implemented, such as connecting the output of the node to the inputs of other nodes on the same layer or connecting the output of the network to the input of the network. A recurrent network introduces other problems, such as a more complex learning rule, slower convergence, and possible instability in the learning process. Useful neural network models using just the input and output data for control purposes are described in [4]. The neural network in this case will act as an observer or plant estimator and it will need to have available the inputs and outputs of the plant. To make a neural network a dynamical plant estimator, dynamical structures based on typical linear system identification models can be used. For example, the neural network input could be augmented with

-past values of the plant's input, or

-past values of the plant's input and output, or

-past values of the predicted outputs and plant's inputs, or

-past values of the predicted outputs and the plant's input and output.

These four structures lead to neural network implementations of standard Finite Impulse Response (FIR), ARX, Output-Error (OE), and ARMAX linear models, respectively.

The choice of the model depends on the complexity of the plant to be modeled. When the plant is stable a FIR model may be used. This model will require a tapped time delay element for each increment of time for the length of the transient response. If the plant has both low and high frequency modes this could require a large number of tapped time delays. This would make this model computationally slow. The ARX model can be used with both stable and unstable plants with little to no output sensor noise. This model requires far less tapped time delays than the FIR model. When sensor noise is present the use of the OE model would produce better results [23]. The ARMAX model is used to model noise whereas OE makes a better model than ARX but does not model noise. In the rest of the thesis the ARX model will be used to model the plant, and a combination of the ARX and OE model will be used for plant prediction. The assumption of little to no noise will be used. These models will be discussed in more detail in the following three sections.

2.4.4.2. Tapped Time Delay Network Architecture for Modeling a Dynamical Plant

To capture the dynamics of a plant, a tapped time delayed network structure with a single hidden layer is used (see Figure 4)[2]. The input to the network can be decomposed into two parts. The first part is the input to the plant and its past values,

[2] Note: To simplify the schematics, an unshaded circle will be used to represent a node as in Figure 1.

where the current time input is denoted as $u(n)$ and n represents the current discrete time. The second part is composed of the plant's previous values, starting with $y(n\text{-}1)$, the previous output. The previous values represent delayed time values stored in delay nodes, and thus, the name, "Time Delay Network". Using this notation the input vector is redefined as follows

$$\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n_d+1} \\ u_{n_d+2} \\ u_{n_d+3} \\ \vdots \\ u_{d_d+n_d+1} \end{bmatrix} \equiv \begin{bmatrix} u(n) \\ u(n-1) \\ \vdots \\ u(n-n_d) \\ y(n-1) \\ y(n-2) \\ \vdots \\ y(n-d_d) \end{bmatrix}, \tag{10}$$

where n_d is the number of past input values, and d_d is the number of past outputs of the plant.

For this development the network will have one hidden layer containing several hidden nodes that use a general activation function f. A single layer was chosen because it has been shown that these neural networks can approximate a measurable set to any desired degree of accuracy [8]. The output node uses a linear output function for scaling the network. This function has a slope of one.

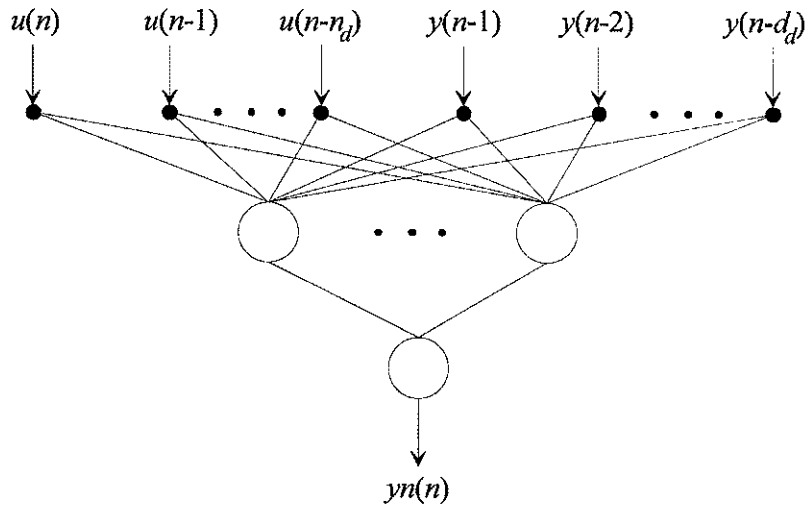

Figure 4. Network Architecture for Modeling a Dynamical Plant

The equation for this network architecture is:

$$net_j(n) = \sum_{i=0}^{n_d} \left\{ w_{j,i+1}\, u(n-i) \right\} + \sum_{i=1}^{d_d} \left\{ w_{j,i+n_d+1}\, y(n-i) \right\} + b_j \tag{11}$$

and

$$yn(n) = \sum_{j=1}^{h_d} \left\{ w_j\, f_j\left(net_j(n) \right) \right\} + b \tag{12}$$

where

$yn(n)$ is the output of the network at time n,

f_j is the activation function for the j^{th} node of the hidden layer,

h_d is the number of hidden nodes in the hidden layer,

w_j the weight connecting the j^{th} hidden node to the output node,

$w_{j,i}$ the weight connecting the i^{th} input node to the j^{th} hidden node,

b_j the bias on the j^{th} hidden node, and

b the bias on the output node.

It will be convenient to represent Figure 4 with a single block representation as shown in Figure 5.

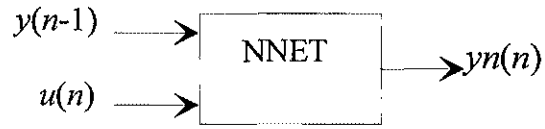

Figure 5. Block Diagram Representation of a Time-Delay Neural Network

The delay nodes for the network are assumed to be contained within the block diagram and are not shown as inputs.

An alternative input may be defined to capture plant dynamics. Using the network output instead of the plants' output for the delayed input converts the static network to a recurrent network. This model may be used for training and is discussed in the next section.

2.4.4.3. Procedure to Train a Neural Network as a Dynamic Plant Estimator

The network is now ready to be placed into its learning environment. The environment for the network in this thesis is a non-linear dynamical plant. The plant is

discretized by placing a digital to analog converter (D/A) at the input of the plant followed by a zero order hold (ZOH) circuit. This signal is then passed to the plant. The output of the plant is then processed through an analog to digital converter (A/D). This process is shown in Figure 6. The objective of the network is to predict the plant's performance for a given input at time $t=nT$, where t is the current time in seconds, n is an integer representing discrete time, and T is the sampling time.

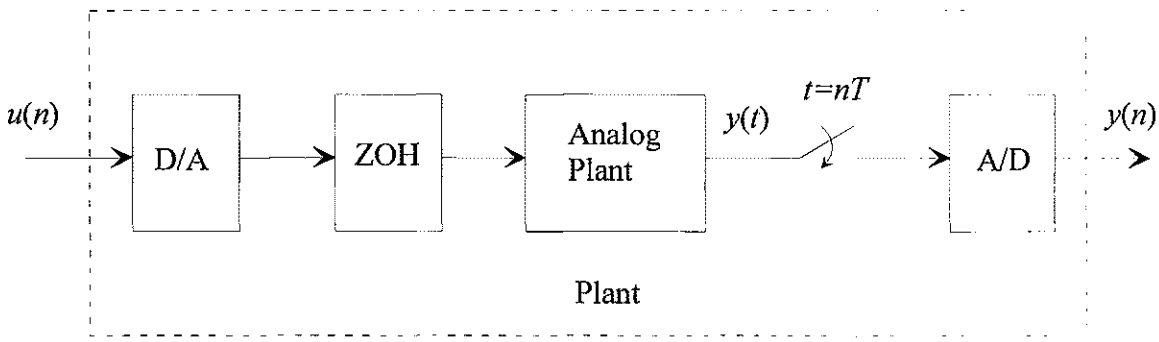

Figure 6. Discrete Plant

The choice of T, the number of tapped time delays n_d and d_d, whether OE or ARX modeling and the excitation signal used is essential for proper modeling of the plant. The sampling time T is typically chosen to be the largest possible number that is sufficient to sample the highest significant modes of the plant. Specifically one should choose T to be about 20 to 40 times the highest mode. To slow sampling will limit the ability to model the high frequency modes. Fast sampling, however, can cause the modeling of a minimum phase plant to be modeled as non-minimum phase plant [31]. There also is a problem with the computer resolution. The faster you sample the less the

difference is on the output of the plant, $y(n)$, between sampling points. This difference determines the magnitude of the weights for the zero dynamics, that is, the weights associated with $u(n)$ and its delays, and the resolution of the weights for the pole dynamics, that is the weights associated with $y(n\text{-}1)$ and its delays. The faster the sampling rate, the smaller the weights are for the zero dynamics and the lower resolution for the pole dynamics.

A typical number of delays n_d and d_d is the order of the nominal plant model plus a few more for unmodeled dynamics. Choosing the number of hidden nodes is still an art. For the types of plants used in the case studies 5 to 8 hidden nodes were very successful.

The choice of whether the plant's output or the network's output is used for the input is of particular importance. The use of the plant's output results in a static network, called the ARX model, and thus the learning algorithm is as defined in Section 2.4.3. The problem with this configuration occurs when there is significant sensor noise. This could introduce bias error in the network parameters (i.e., weights and biases). This a well-known result in system identification using Infinite Impulse Response (IIR) filters [23]. When there is significant sensor noise it is necessary to use the output of the network for learning. This converts the network to a recurrent network, called the OE model, and thus requires a recurrent learning algorithm. Recurrent learning algorithms for a general recurrent network are computationally prohibitive, thus a first order

approximation is typically used [4]. The first order approximation is the same algorithm described in Section 2.4.3. Since this is an approximation, the convergence is slower and less stable. It is recommended that the OE modeling be used only when the ARX model does not produce good results. The thesis assumes the sensor noise is negligible, thus, will use the ARX model for training the network.

The excitation signal for learning the plant's dynamics should span the input space over the region that the plant will be controlled. This could be accomplished with a random signal or a sequence of increasing pulses. The random signal could be used to provide sufficient excitation that could help in the training and generalization capabilities. Our experiments have shown that the pulse train in Figure 7 excites all the modes of the plant as does the random signal. With the increasing amplitude the pulse train excites the nonlinear characteristics of the plant. The pulse train's maximum amplitude is chosen to be the maximum control input to be used. This is typically at the actuator saturation limit. The pulse width of the pulses is chosen so that the plant reaches a steady state value for each pulse. The number of pulses should be chosen so that the network can interpolate the dynamics of the plant. The pulses alternate in sign to excite positive and negative inputs to the plant where needed. There are cases that this would not be reasonable (see case study magnetic levitation, Section 4.7). Based on my experience a reasonable number of pulses are 5 positive and 5 negative pulse for the plants under investigation. Figure 7 shows a typical pulse train.

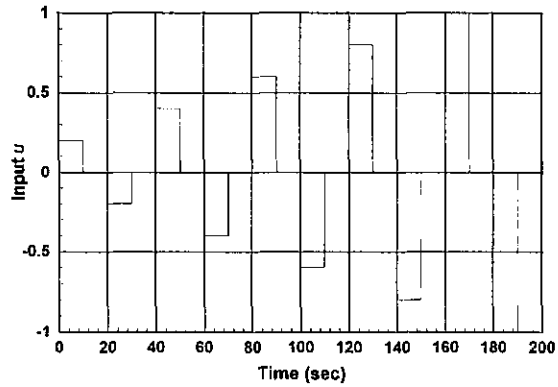

Figure 7. Typical Pulse Train to Excite a Dynamical Plant.

The initialization of the weights is done as specified in Section 2.4.3 except when the network is initialized with a nominal linear plant model. Then a correlation of the weights of the network and a discrete linear model of the plant is needed. This is covered in Section 3.6.1

Training a network to be a dynamical system estimator is accomplished in the same manner as in Section 2.4.3. The training procedure can be described as follows. First, a forward pass through the network processes the input $u(n)$, previous output $y(n\text{-}1)$, and their past values, giving the current output $y(n)$. Second, the error signal is formed by subtracting the neural network's output, $yn(n)$, from the corresponding dynamical plant's output, $y(n)$. Third, the error at the output of the network is backpropagated to form the errors at the hidden layer. Fourth the errors are used to find weight updates using equations (8) and (9). The block diagram of this process is shown in Figure 8.

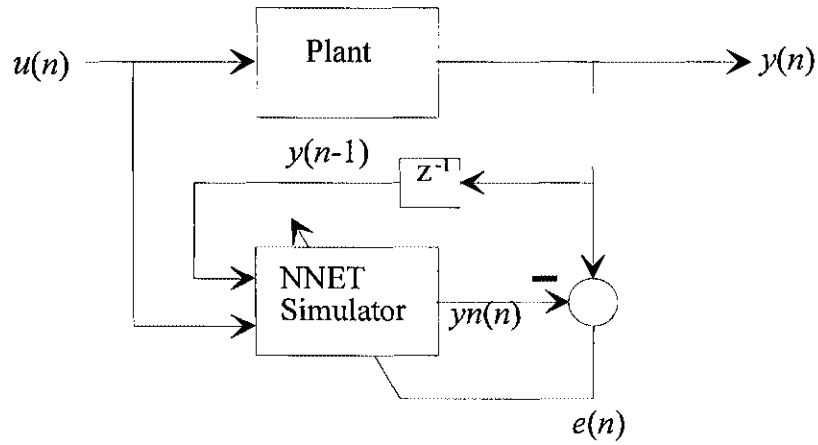

Figure 8. Block Diagram of a Neural Network Wired to Learn the Dynamics of a Plant

2.4.4.4. Neural Network Prediction of Plant Dynamics

Section 2.2.4.3 gives a procedure to train a neural network as a dynamical system estimator that not only depends on current and past values of the plant's input but, also on past values of the plant's output. In Chapter 3, a Neural Generalized Predictive Control method will be presented. This method requires that the trained neural network estimator predicts the performance of the plant for some future time from n+1 to n+k. During the prediction process, neural network predicted outputs replace the plant's outputs. To predict the plant's outputs from the current time, n, to some future time n+k, equation (12) is time shifted by k, and the predicted results are introduced into equation (11) resulting in

$$yn(n+k) = \sum_{j=1}^{h_d} \left\{ w_j \, f_j\left(net_j(n+k)\right) \right\} + b, \tag{13}$$

and

$$net_j(n+k) = \sum_{i=0}^{n_d} w_{j,i+1} u(n+k-i) + \sum_{i=1}^{\min(k,d_d)} \left(w_{j,n_d+i+1}\, yn(n+k-i) \right) + \sum_{i=k+1}^{d_d} \left(w_{j,n_d+i+1}\, y(n+k-i) \right) + b_j \quad (14)$$

The second and third summations in (14) introduce the predicted outputs, feeding back the network output, *yn*, for k or d_d times, whichever is smaller. The last summation of (14) uses the previous available values of the plant's output, *y*, whenever $k<d_d$.

Example

Consider a network with input nodes consisting of $u(n)$, its' two previous inputs, (i.e., n_d=2), and the three previous outputs (i.e., d_d=3); two hidden nodes (i.e., h_d=2); and one output node. Suppose that a 2-step prediction needs to be found, that is, the network needs to predict the output at times n+1 and n+2. The information required by the neural network at each time instant is given in Figure 9.

The example below depicts a prediction of the plant for k=2. To produce the output $yn(n+2)$, the inputs $u(n+1)$ and $u(n+2)$ are needed. The prediction process is started at time n, with the initial conditions of $[u(n)\ u(n-1)]^T$ and $[y(n)\ y(n-1)\ y(n-2)]^T$ and the estimated input $u(n+1)$. The output of this process is $yn(n+1)$, which is fed back to the network and the process is repeated to produce the predicted plant's output $yn(n+2)$. This

process is shown in Figure 9 where the network feedback is shown as one network feeding another.

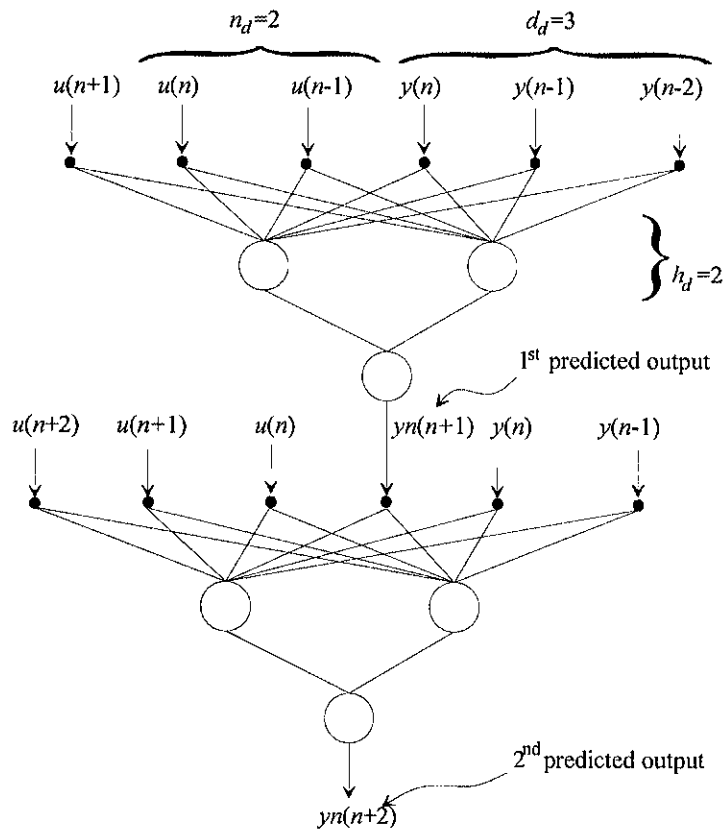

Figure 9. Network Prediction for k=2

2.4.4.5. Measures of Estimation Performance

Once the network is trained, a way to determine how well the network performs as an estimator needs to be determined. The standard RMS measure is fine, but a good RMS value will vary from problem to problem. It is desirable to have a measure that is

invariant over the problem being learned. This is accomplished by normalizing the RMS measure with respect to the plant's output, resulting in

$$\mathbf{NRMS} = \sqrt{\frac{\sum_{n}\left(y(n)-yn(n)\right)^2}{\sum_{n} y(n)^2}}\ , \tag{15}$$

where the summations are made over the time interval of interest. The response of the network is obtained by a using the recurrent configuration, that is the OE model. The use of the recurrent network for the NRMS will indicate whether there is bias error due to sensor noise (discussed in Section 2.4.4.3). The error bias is determined by the difference between the ARX and OE NRMS measures. A good OE NRMS error is an indication that the network has learned the plant dynamics. From various experiments it has been observed the an acceptable NRMS value is 10^{-2}, a good value is 10^{-3}, and a very good value is 10^{-4}.

Another measure that is used is the maximum error (Max Error) between $y(n)$ and $yn(n)$ over the time interval of interest. This error measures the worst case error the network will produce.

2.5. Neural Network Control Systems

2.5.1. Introduction

Control algorithms can be classified by the approaches used. The class of approaches used in control algorithms are classical, expert systems, fuzzy-logic and neural networks (see Figure 10). The classical approach includes linear and nonlinear controls such as, PID, LQR, feedback linearization, etc.. Expert systems and fuzzy-logic are rule based approaches that can learn their rules by experience as in a neural network. Each of these approaches is partitioned into hybrid and pure approaches. Hybrid approaches are combinations of any or all of the four approaches, for example, Kawato's Classical-Neural controller [35,36]. Pure approaches use only the elements from that technique. The following literature search will focus on pure neural network control algorithms.

2.5.2. Short Literature Search

Pure neural network control algorithms are decomposed into Reinforcement and Supervised approaches [37]. The Adaptive Critic consists of two neural networks. The first network is the controller. The second network is a critic for the neural control. The critic informs the neural control on the system performance for adaptation. Barto et al. demonstrated this approach to control an inverted pendulum [34]. Detailed explanations of this approach can be found in [2] and [3] by Barto. The Supervised approach utilizes the difference between a reference signal and the plant output to determine the next

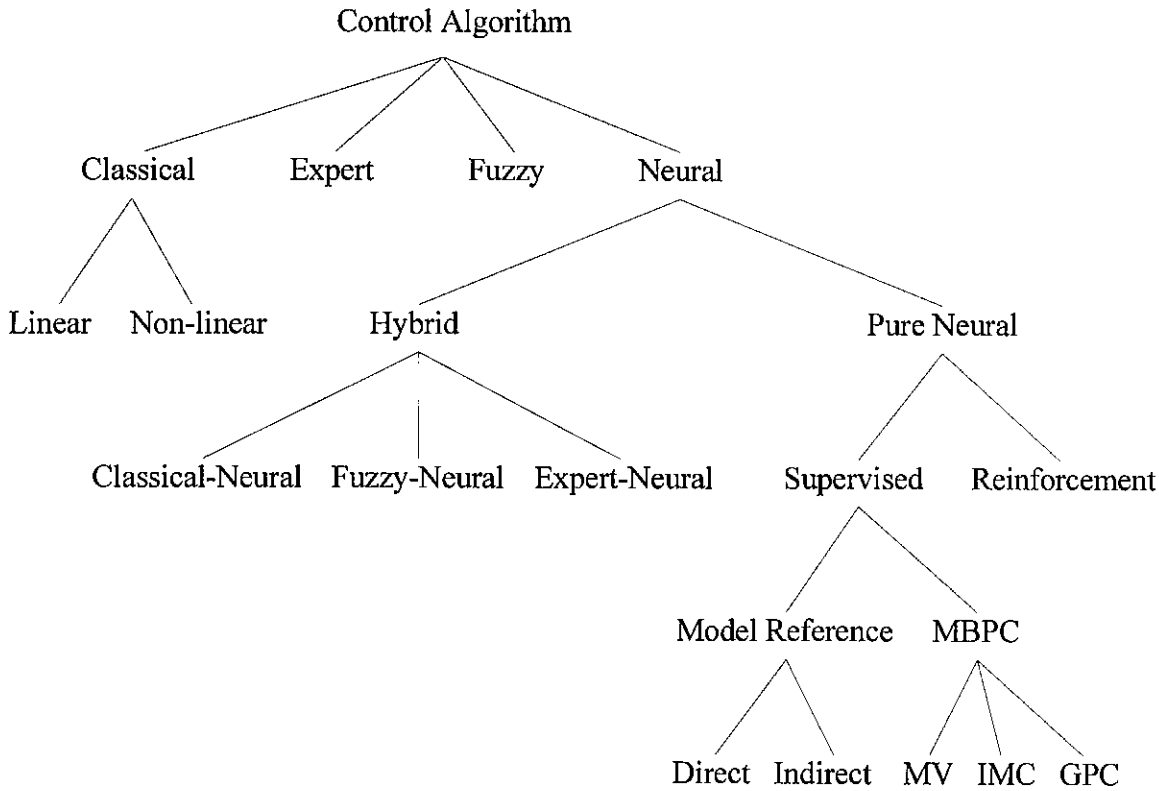

Figure 10. Control Algorithm Tree

control input. The signal used to update the weights decomposes the Supervised approach into two classes. Model Reference uses the difference between a reference signal and the plant output to update the weights. The approach that uses this signal directly to update the weight is called the Direct approach [38]. Were as if this signal is processed through some device before weight update, this approach is call Indirect [40]. The model reference approach trains a network to learn the inverse or a filtered inverse of the plant. Model Based Predictive Control (MBPC) uses the difference between the plant and the neural model to adapt the neural network to learn a forward model of the plant. MBPC minimizes a cost function to determine the control input. The choice of the cost function differentiates between the type of MBPC used. Several MBPC techniques are Minimum Variance [32,33], Internal Model Control [24], and Generalized Predictive Control [41]. These three algorithms follow the linear counterparts in their approach and cost function. The neural approach differs from the linear approach in the manner the cost function is solved. Nguyen and Widrow demonstrate a Minimum Variance controller performing the truck backer upper problem [33]. Hunt and Sbarbaro develop and show the connection between inverse control and Internal Model Control. They follow with the development of Neural Internal Model Control [24]. A detailed discussion of Neural Generalized predictive control is given in Section 3.4.2.

3. Development of Neural Generalized Predictive Control

3.1. Introduction

This chapter presents a neural network control design method based on Generalized Predictive Control (GPC) where the predictor model is replaced with a neural network. This is called Neural Generalized Predictive Control (NGPC) method. The standard cost function of GPC is used and we derive a Newton-Raphson implementation of the cost function minimization (CFM) algorithm. This CFM algorithm makes it possible to use NGPC in real-time control applications. The CFM algorithm is then augmented to handle actuator constraints that result on amplitude constraints for the plant input.

3.2. Organization of Chapter

In Section 3.3 Generalized Predictive Control is introduced. In Section 3.4 previous work in Neural Generalized Predictive Control is reviewed, the algorithm is discussed and the cost function with actuator constraints is defined. In Section 3.5 the Newton-Raphson approach to minimizing the cost function is derived. Section 3.6 discuses procedures for real-time control and presents the relationship between the

network weights and linear difference equations. Section 3.7 concludes with a summary of the main contribution of this thesis.

3.3. Generalized Predictive Control (GPC)

Generalized Predictive Control (GPC) belongs to a class of digital control methods called Model-Based Predictive Control (MBPC) [20]. MBPC techniques have been analyzed and implemented successfully in process control industries since the end of the 1970's [25]. A recent bibliography was compiled by Clarke in [25]. Clarke and coworkers introduced GPC in 1987 [20,21]. MBPC techniques continue to be used in process control because they can systematically take into account real plant constraints in real-time. In particular, GPC can control non-minimum phase plants, open-loop unstable plants and plants with variable or unknown dead time.

The basic structure of MBPC methods consists of a predictor model and an optimization algorithm that minimizes a particular cost function. The choice of different prediction models and optimization algorithms lead to different MBPC techniques. Initially, the prediction models were simply generated from step response values at the sampling instants. For GPC, a model that takes into account the effect of noise is used. The model is usually called Controlled Auto-Regressive Integrated Moving-Average (CARIMA) model which in standard terminology would be called AutoRegressive

Integrated Moving Average eXogeneous input (ARIMAX) model. This input/output model is given by:

$$A(q)y(t) = B(q)u(t) + C(q)e(t),$$

where q denotes the shift operator, and $A(q)$, $B(q)$, and $C(q)$ are polynomials in q^{-1}. The A and B polynomial define the plant's dynamics, that is the poles and zeros, respectively. Typically in linear systems the b_0 coefficient in the B polynomial is set to zero. In the model used here the b_0 term will be learned. The *C/A* transfer function characterizes the response to sensor noise and external disturbances.

GPC is an MBPC technique that uses a long range prediction horizon cost function. At each sampling instant GPC uses predicted values from the predictor model to minimize a cost function that takes into account predicted tracking errors and the control signal. Part of the success of these techniques is due to the fact that they do not employ a 1-step ahead predictor but rather an N_2-step ahead predictor where N_2 is greater than 1 and finite. An appropriate choice for N_2 and other design parameters leads to good control properties.

The GPC cost function presented in [20] is given by

$$J = \sum_{j=N_1}^{N_2} \left[ym(n+j) - yn(n+j) \right]^2 + \sum_{j=1}^{N_u} \lambda(j) \left[\Delta u(n+j) \right]^2 \qquad (16)$$

where

N_1 is the minimum costing horizon,

N_2 is the maximum costing horizon,

$\lambda(j)$ is a control-weighting sequence,

ym is the reference trajectory,

yn is the prediction of the plant's performance, and

$\Delta u(n)$ is the incremental change of the control input u at time n.

It consists of two parts. The first part minimizes the predicted tracking error, the difference between the reference model, *ym*, and the predicted output of the plant model. This minimization is traded-off with the second part that minimizes the control energy of the sequence of predicted inputs, $\{u(n+1), \ldots, u(n+N_2)\}$. A weighting factor, $\lambda(j)$, is introduced to control the balance between the two parts. The weighting factor acts as a damper on the predicted u. N_1 different from 1 is used when the plant has dead time. It is set to one plus the dead time of the plant. The choice of $N_1>1$ would improve the computational cost, but cannot be greater than one plus the dead time, because plant performance would degrade. N_2 is chosen as the output prediction horizon. That is, how far into the future GPC predicts the performance of the plant. Clarke shows that an N_2 of 10 is sufficient for most systems [21], but N_2 is not necessarily as high as ten. Another choice for N_2 is to make the predictions until the expected settling time of the plant. There are several rules of thumb to select these parameters but no systematic technique

exists. The basic rules that we found useful to modify the parameters are given in Section 3.6.

Since the cost function is quadratic, analytic solutions of the minimization are possible using one of the prediction models. The analytic solutions lead to a standard implementation of a feedback digital controller. In practice, an advantage of MBPC techniques is that they can also handle implementation constraints such as limits on actuator signal amplitudes. In this case, the cost function is minimized on-line. The slow time constants of process control plants have made it possible for a variety of numerical techniques to be used in real-time control.

An on-line GPC algorithm could be implemented as follows:

1. Generate a reference trajectory $\left[ym(n+N_1)\ \ ym(n+N_1+1)\cdots ym(n+N_2)\right]^{\mathrm{T}}$. If the future trajectory of $ym(n)$ is unknown, keep $ym(n)$ constant for the future trajectory. For real-time minimization of the cost function, the reference trajectory is usually smoothed using a reference model.
2. Start with the previously calculated control input vector, $\left[u(n+1)\ \ u(n+2)\cdots u(n+N_2)\right]^{\mathrm{T}}$. If it is the first time though the algorithm, start the control input vector equal to the zero vector. With this input generate a predicted output vector of the plant, $\left[yn(n+N_1)\ \ yn(n+N_1+1)\cdots yn(n+N_2)\right]^{\mathrm{T}}$, using the plant model.

3. Calculate a new sequence of control inputs, $\left[u(n+1)\ \ u(n+2)\cdots u(n+N_2)\right]^{\mathrm{T}}$, that minimizes the cost function,
4. Repeat steps 2 and 3 until the desired minimization is achieved,
5. Send the first control input to the plant,
6. Repeat the entire process for each time step.

Figure 11 shows a block diagram of the above process. The Cost function Minimization (CFM) algorithm evaluates steps 3 and 4. The double pole double throw switch, S, is in the down position when doing prediction. Having this switch lets the plant model use previous measured plant outputs for future predictions and in the down position prediction of the plant model can be performed.

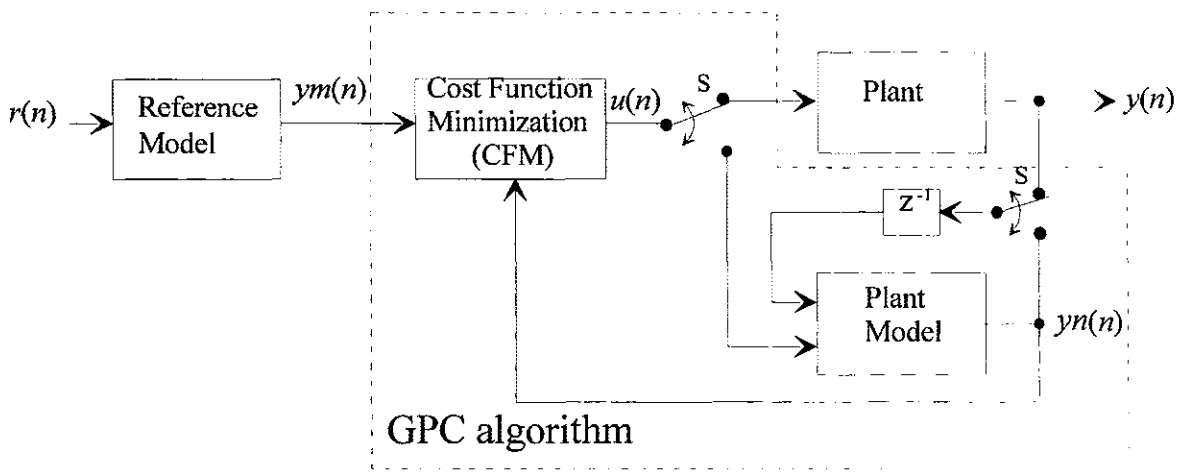

Figure 11. GPC Block Diagram

The standard GPC algorithm is limited by the use of a linear predictor model. One way that GPC can handle nonlinear plants is to linearize the plant about a set of operating points. If the plant is highly nonlinear the set of operating points can be very large. Another technique involves developing a nonlinear model which depends on making assumptions about the dynamics of the nonlinear plant. If these assumptions are incorrect the accuracy of the model will be reduced. Neural networks have been shown to be good nonlinear dynamical system estimators [4]. By using a neural network as the predictor for GPC the ability of the controller to make more accurate predictions about the nonlinear plant are improved. The next section describes a technique to use neural networks with GPC.

3.4. Neural Generalized Predictive Control

3.4.1. Introduction

To use GPC in the control of nonlinear plants, a nonlinear black box estimator is needed. This black box model should be able to estimate the nonlinear dynamics on-line. A class of nonlinear black-box model estimators that meet these requirements are neural networks. Several papers have already proposed the use of neural networks as plant estimators. Using a neural network as the predictor for GPC improves the ability to make more accurate predictions for control of the nonlinear plant. Improved predictions affect stability and performance margins, rise time, over-shoot, and the energy contained in the control signal.

3.4.2. Previous Work

GPC had been originally developed with linear plant models. The quadratic cost function with linear plant models leads to analytic solutions. With the use of neural networks for the plant model, the linear systems approach breaks down, and thus a nonlinear optimization algorithm is necessary. Several researchers have realized the benefits of merging neural network modeling dynamical systems and GPC. This approach to GPC requires the minimization of a nonlinear cost function. These researchers have tried various optimization techniques such as Non-gradient [10], Simplex [12], Successive Quadratic Programming [9,11], and others. Of primary concern in any approach is consideration that computational expense limits practical use. In general, these approaches are computationally costly thus making real-time control difficult. Very few papers address real-time implementation, or they have plants that have a large time constant [26], and [27]. Koivisto [28] addresses the convergence rate for the optimization algorithm. To improve the usability, a faster optimization algorithm is needed. None of the previous implementations use Newton-Raphson as an optimization technique. Newton-Raphson is a quadratically converging algorithm while the others have less than a quadratic convergence. The improved convergence rate of Newton-Raphson is computationally costly, but is justified by the high convergence rate of Newton-Raphson (see Sections 4.3 and 4.4).

3.4.3. NGPC Basic Algorithm

The NGPC algorithm is the same as the GPC algorithm but the linear predictor model is replaced with a neural network. To use a neural network as a plant predictor, a training procedure needs to be established. We will use the techniques presented in Section 2.4.4.3. The neural network is integrated with the cost function and a cost function minimization is developed. The cost function is the same basic cost function for the GPC algorithm.

The NGPC system consists of four components, the plant to be controlled, a reference model that specifies the desired performance of the plant, a neural network that models the plant, and the Cost Function Minimization (CFM) algorithm that determines the input needed to produce the plant's desired performance. The NGPC algorithm consists of the CFM block and the neural network block. Figure 12 shows the block diagram of the NGPC system.

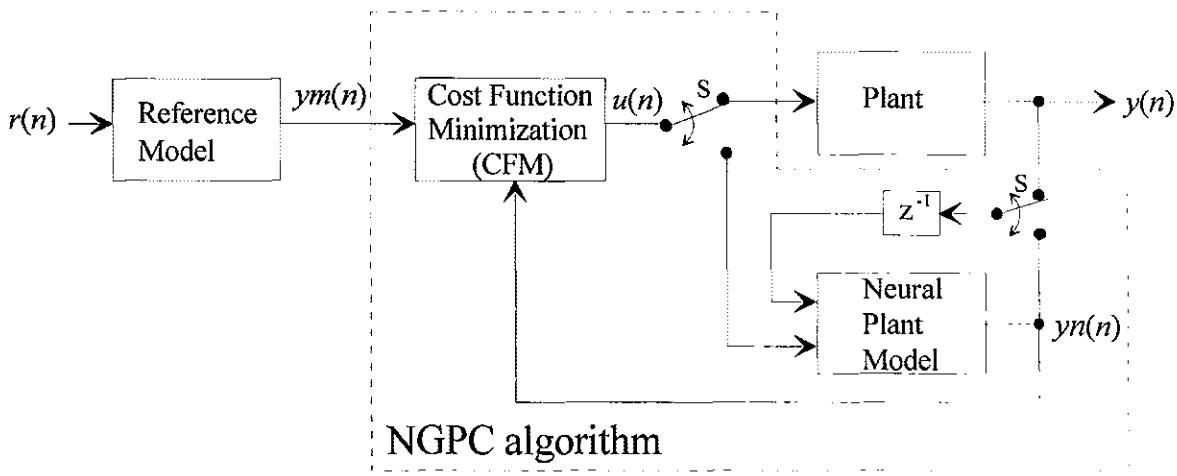

Figure 12. Block Diagram of NGPC

The NGPC system starts with the input signal, $r(n)$, which is presented to the reference model. This model produces a tracking reference signal, $ym(n)$, that is used as an input to the CFM block. The CFM algorithm produces an output which is either used as an input to the plant or the plant's model. The double pole double throw switch, S, is set to the plant when the CFM algorithm has solved for the best input, $u(n)$, that will minimize the specified cost function. Between samples, the switch is set to the plant's neural network model where the CFM algorithm uses this model to calculate the next control input, $u(n+1)$, from predictions of the model's response. Once the cost function is minimized, this input is passed to the plant and this process is repeated

There are several additions that can be used with the basic cost function (16). One of these is to constrain the control input to some range. This is important with plants that have actuator saturation. This is considered in the next section.

3.4.4. NGPC Cost Function with Actuator Constraints

An advantage of GPC and NGPC is that real plant constraints can be easily taken into account in the cost function. An important common actuator constraint limits the amplitude of the control signal. Adding the input constraint function to the basic cost function results in

$$J = \sum_{j=N_1}^{N_2}\left[ym(n+j) - yn(n+j)\right]^2 + \sum_{j=1}^{N_2}\lambda(j)\left[\Delta u(n+j)\right]^2 + \sum_{j=1}^{N_2}\left[\frac{s}{u(n+j)+\frac{r}{2}-c} + \frac{s}{\frac{r}{2}+c-u(n+j)} - \frac{4}{r}\right] \tag{17}$$

where

$\Delta u(n+j) = u(n+j) - u(n-1+j)$,

N_1 is the minimum costing horizon,

N_2 is the maximum costing horizon,

$\lambda(j)$ is a control-weighting sequence,

s is the sharpness of the constraint function,

r is the range for the constraint, and

c is the center of the range for the constraint.

N_1 specifies the dead time of the plant and N_2 is the horizon. This cost function has three parts. The first part minimizes the error between the model, $ym(n)$, and the neural network, $yn(n)$. The second part minimizes the rate of change for the inputs, $u(n+j)$, with λ as this constraint's weight. The scalar λ acts as a damping of the control input $u(n+1)$. The third part is an input constraint. This will constrain the control input to the range $c-\frac{r}{2} \le u \le c+\frac{r}{2}$. The input constraint will guarantee that the control input will not saturate the actuator. The parameters s, r, and c characterize the sharpness, range, and center of the input constraint function respectively. The sharpness, s, controls

the shape of the constraint function. This will be discussed in more detail is Section 3.5.4. The $4/r$ causes the constraint function to equal zero when the control input is equal to the center. This term has no affect on the solution but allows for proper evaluation of J.

The technique used to minimize the cost function is the key to real-time control. Many techniques can be used but these techniques prove to be slow. The next section develops a real-time minimization based on the Newton-Raphson optimization algorithm.

3.5. Cost Function Minimization (CFM)

3.5.1. Introduction

Section 3.3 described the GPC algorithm with the basic cost function (16). The minimization of the basic cost function with the plant model with an ARIMAX model, can be calculated analytically using the Diophantine equation [20]. When the plant model is a neural network, linear techniques will not work. Therefore the use of a nonlinear iterative solution is necessary. These techniques can be divided into two cases: gradient techniques, and non-gradient techniques. Only one of the gradient techniques is a quadratically converging algorithm. This algorithm, Newton-Raphson, is the fastest converging algorithm when measured in terms of iterations. This algorithm could be faster if the parameters are computationally inexpensive to calculate. It is critical for real-time control that the CFM be calculated efficiently. Presented here, is a Newton-Raphson

solution to the augmented cost function that is about an order of magnitude more efficient per solution than a first order gradient based algorithm.

A review of the Newton-Raphson algorithm is first presented in Section 3.5.2. It is followed by the development of the Newton-Raphson equation for the NGPC cost function in Section 3.5.3. Section 3.4 is concluded with a detailed description of the minimization of the actuator constraint function in Section 3.5.4.

3.5.2. Review of Newton-Raphson Algorithm

Newton-Raphson is a quadratically converging optimization technique for solving nonlinear equations of the form $f(\mathbf{x})=0$, where $f(\mathbf{x})$ is a nonlinear differentiable function in the space of $\mathbf{x}$, and $\mathbf{x} \in \Re^n$.

The derivation of the Newton-Raphson method is as follows:

Start by taking a guess for the solution $\mathbf{x}$, say $\mathbf{x}_n$. To improve the guess we first expanding $f(\mathbf{x})$ in a Taylor series about $\mathbf{x}=\mathbf{x}_n$ for the improved guess, $\mathbf{x}_{n+1}$, resulting in

$$f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n) + f'(\mathbf{x}_n)(\Delta\mathbf{x}) + \frac{1}{2!}f''(\mathbf{x}_n)(\Delta\mathbf{x})^2 + \frac{1}{3!}f'''(\mathbf{x}_n)(\Delta\mathbf{x})^3 + \cdots \tag{18}$$

where

$$\Delta\mathbf{x} = \mathbf{x}_{n+1} - \mathbf{x}_n . \tag{19}$$

Taking the first two terms, $f(\mathbf{x}_{n+1}) \cong f(\mathbf{x}_n) + f'(\mathbf{x}_n)(\Delta\mathbf{x})$, and assuming that $f(\mathbf{x}_{n+1})=0$ as desired, we solve for $\Delta\mathbf{x}$ and obtain $\Delta\mathbf{x} = -\frac{f(\mathbf{x}_n)}{f'(\mathbf{x}_n)}$. Substituting this into (19) we find the next solution $\mathbf{x}_{n+1} = \mathbf{x}_n - \frac{f(\mathbf{x}_n)}{f'(\mathbf{x}_n)}$.

To find the local extrema of a function, say $g(\mathbf{x})$, we take the first derivative of $g(\mathbf{x})$ with respect to $\mathbf{x}$ and set it equal to zero, that is

$$\frac{dg(\mathbf{x})}{d\mathbf{x}} = 0,$$

thus the update rule for the extrema is

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \frac{g'(\mathbf{x}_n)}{g''(\mathbf{x}_n)}.$$

Here we see that when dealing with vector functions the Jacobian and the Hessian are calculated for each iteration. It is shown in Section 4.3 that many of the terms for the calculation of the Hessian have been calculated for the Jacobian. Other gradient approaches require at least the calculation of the Jacobian. Since some of the Hessian is calculated from the Jacobian this approach is faster than other gradient based techniques because of its convergence rate with minimal additional computational cost

3.5.3. CFM Algorithm

The objective of the CFM algorithm is to minimize J in (17) with respect to the vector $[u(n+1), u(n+2), \ldots, u(n+N_2)]^T$, denoted $\mathbf{U}$. This is accomplished by setting the

Jacobian of (17) to zero and solving for **U**. With Newton Raphson used as the CFM algorithm, J is minimized iteratively to determine the best **U**. An iterative process yields intermediate values for J denoted $J(k)$. For each iteration of $J(k)$ an intermediate control input vector is also generated and is denoted as

$$\mathbf{U}(k) \stackrel{\Delta}{=} \begin{bmatrix} u(n+1) \\ u(n+2) \\ \vdots \\ u(n+N_2) \end{bmatrix}, \quad k=1,\ldots,\#\text{iterations}.$$

The Newton-Raphson update rule for $\mathbf{U}(k+1)$ is

$$\mathbf{U}(k+1) = \mathbf{U}(k) - \left(\frac{\partial^2 \mathbf{J}}{\partial \mathbf{U}^2}(k) \right)^{-1} \frac{\partial \mathbf{J}}{\partial \mathbf{U}}(k), \tag{20}$$

where the Jacobian is denoted as

$$\frac{\partial \mathbf{J}}{\partial \mathbf{U}}(k) = \begin{bmatrix} \dfrac{\partial J}{\partial u(n+1)} \\ \vdots \\ \dfrac{\partial J}{\partial u(n+N_2)} \end{bmatrix}$$

and the Hessian as

$$\frac{\partial^2 \mathbf{J}}{\partial \mathbf{U}^2}(k) = \begin{bmatrix} \dfrac{\partial^2 J}{\partial u(n+1)^2} & \cdots & \dfrac{\partial^2 J}{\partial u(n+1)\partial u(n+N_2)} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 J}{\partial u(n+N_2)\partial u(n+1)} & \cdots & \dfrac{\partial^2 J}{\partial u(n+N_2)^2} \end{bmatrix}.$$

Solving Equation (20) directly requires the inverse of the Hessian matrix. This processes could be computationally expensive. One technique to avoid the use of a

matrix inverse is to use LU decomposition [22] to solve for the control input vector $\mathbf{U}(k+1)$. This is accomplished by rewriting Equation (20) in the form of a system of linear equations, $\mathbf{Ax} = \mathbf{b}$. This results in

$$\frac{\partial^2 \mathbf{J}}{\partial \mathbf{U}^2}(k)\left(\mathbf{U}(k+1) - \mathbf{U}(k)\right) = -\frac{\partial \mathbf{J}}{\partial \mathbf{U}}(k), \tag{21}$$

where

$$\frac{\partial^2 \mathbf{J}}{\partial \mathbf{U}^2}(k) = \mathbf{A},$$

$$-\frac{\partial \mathbf{J}}{\partial \mathbf{U}}(k) = \mathbf{b}, \text{ and}$$

$$\mathbf{U}(k+1) - \mathbf{U}(k) = \mathbf{x}.$$

In this form Equation (21) can be solved with two routines supplied in [22]. That is the LU decomposition routine, ludcmp, and the system of linear equations solver lubksb.

After $\mathbf{x}$ is calculated, $\mathbf{U}(k+1)$ is solved by evaluating $\mathbf{U}(k+1) = \mathbf{x} + \mathbf{U}(k)$. This procedure is repeated until the percent change in each element of $\mathbf{U}(k+1)$ is less than some ε. When solving for $\mathbf{x}$, calculation of each element of the Jacobian and Hessian is needed for each Newton Raphson iteration. The h^{th} element of the Jacobian is

$$\frac{\partial J}{\partial u(n+h)} = -2\sum_{j=N_1}^{N_2}\left[ym(n+j)-yn(n+j)\right]\frac{\partial yn(n+j)}{\partial u(n+h)} + 2\sum_{j=1}^{N_2}\lambda(j)\left[\Delta u(n+j)\right]\frac{\partial \Delta u(n+j)}{\partial u(n+h)}$$

$$+\sum_{j=1}^{N_2}\delta(h,j)\left[\frac{-s}{\left(u(n+j)+\frac{r}{2}-b\right)^2}+\frac{s}{\left(\frac{r}{2}+b-u(n+j)\right)^2}\right] \quad ,h=1,\ldots,N_2.$$

The $\frac{\partial \Delta u(n+j)}{\partial u(n+h)}$ when expanded and evaluated can be rewritten in terms of the Kronecker delta function[3],

$$\frac{\partial u(n+j)}{\partial u(n+h)} - \frac{\partial u(n+j-1)}{\partial u(n+h)} = \delta(h,j)-\delta(h,j-1).$$

The m^{th}, h^{th} element of the Hessian is

$$\frac{\partial^2 J}{\partial u(n+m)\partial u(n+h)} = 2\sum_{j=N_1}^{N_2}\left\{\frac{\partial yn(n+j)}{\partial u(n+m)}\frac{\partial yn(n+j)}{\partial u(n+h)} - \frac{\partial^2 yn(n+j)}{\partial u(n+m)\partial u(n+h)}\left[ym(n+j)-yn(n+j)\right]\right\}$$

$$+2\sum_{j=1}^{N_2}\lambda(j)\left\{\frac{\partial \Delta u(n+j)}{\partial u(n+m)}\frac{\partial \Delta u(n+j)}{\partial u(n+h)} + \Delta u(n+j)\frac{\partial^2 \Delta u(n+j)}{\partial u(n+m)\partial u(n+h)}\right\}$$

$$+\sum_{j=1}^{N_2}\delta(h,j)\delta(m,j)\left[\frac{2s}{\left(u(n+j)+\frac{r}{2}-b\right)^3}+\frac{2s}{\left(\frac{r}{2}+b-u(n+j)\right)^3}\right] \quad \begin{matrix} ,h=1,\ldots,N_2 \\ ,m=1,\ldots,N_2 \end{matrix}.$$

Again, the delta notation can be used to express

$$\frac{\partial \Delta u(n+j)}{\partial u(n+h)}\frac{\partial \Delta u(n+j)}{\partial u(n+m)} = \left(\delta(h,j)-\delta(h,j-1)\right)\left(\delta(m,j)-\delta(m,j-1)\right).$$

The $\frac{\partial^2 \Delta u(n+j)}{\partial u(n+m)\partial u(n+h)}$ always evaluates to zero.

---

[3] The Kronecker Delta function is defined as $\delta(h,j)=\begin{cases}1 & \text{if } h=j \\ 0 & \text{if } h\neq j\end{cases}$.

To evaluate the Jacobian and the Hessian the network's first and second derivative with respect to the control input vector are needed. The elements of the Jacobian are obtained by differentiating $yn(n+k)$ in Equation (13) with respect to $u(n+h)$ resulting in

$$\frac{\partial yn(n+k)}{\partial u(n+h)} = \sum_{j=1}^{h_d} w_j \frac{\partial f_j\left(net_j(n+k)\right)}{\partial u(n+h)}. \tag{22}$$

Applying the chain rule to $\partial f_j\left(net_j(n+k)\right)/\partial u(n+h)$ results in

$$\frac{\partial f_j\left(net_j(n+k)\right)}{\partial u(n+h)} = \frac{\partial f_j\left(net_j(n+k)\right)}{\partial net_j(n+k)} \frac{\partial net_j(n+k)}{\partial u(n+h)}, \tag{23}$$

where $\partial f_j\left(net_j(n+k)\right)/\partial net_j(n+k)$ is the output function's derivative and

$$\frac{\partial net_j(n+k)}{\partial u(n+h)} = \sum_{i=0}^{n_d} w_{j,i+1}\delta(k-i,h) + \sum_{i=1}^{\min(k-1,d_d)} w_{j,i+n_d+1} \frac{\partial yn(n+k-i)}{\partial u(n+h)}. \tag{24}$$

The elements of the Hessian are obtained by differentiating (22),(23), and (24) with respect to $u(n+m)$, resulting in

$$\frac{\partial^2 yn(n+k)}{\partial u(n+h)\partial u(n+m)} = \sum_{j=1}^{h_d} w_j \frac{\partial^2 f_j\left(net_j(n+k)\right)}{\partial u(n+h)\partial u(n+m)}, \tag{25}$$

$$\begin{aligned}\frac{\partial^2 f_j\left(net_j(n+k)\right)}{\partial u(n+h)\partial u(n+m)} = & \frac{\partial f_j\left(net_j(n+k)\right)}{net_j(n+k)} \frac{\partial^2 net_j(n+k)}{\partial u(n+h)\partial u(n+m)} \\ & + \frac{\partial^2 f_j\left(net_j(n+k)\right)}{net_j(n+k)^2} \frac{\partial net_j(n+k)}{\partial u(n+h)} \frac{\partial net_j(n+k)}{\partial u(n+m)},\end{aligned} \tag{26}$$

and

$$\frac{\partial^2 net_j(n+k)}{\partial u(n+h)\partial u(n+m)} = \sum_{i=1}^{\min(k-1,d_d)} w_{j,mm+i} \frac{\partial^2 yn(n+k-i)}{\partial u(n+h)\partial u(n+m)}.$$

Note that Equation (26) is the result of also applying the chain rule twice.

3.5.4. Analysis of the Actuator Constraint Function Minimization

One of the issues for the controller is how to handle the constraint on the actuators input. A constraint on an input would be the voltage and/or current that could be supplied to an electro-mechanical device like a robotic joint. Typically, hard constraints are used for actuator input constraint function. This is a saturation function that is not differentiable. The Newton-Raphson algorithm requires that the function to be minimized be differentiable. Here we present a differentiable function within the constraint range that approximates the hard constraint to an arbitrary accuracy. The input constraint function is defined by,

$$g(u) = s\left(\frac{1}{u+\frac{r}{2}-c}+\frac{1}{\frac{r}{2}+c-u}-\frac{4}{r}\right), \tag{27}$$

where u is the input variable to be constrained, r is the range of the constraint, and c is the center of the constraint range. The function was designed to produce an infinite cost at the constraint limits, a negligible cost elsewhere, and at the center of the constraint function zero cost. The parameters r, c, and s are used to adjust the range, center, and sharpness of the constraint function respectively. When plotted, the constraint function looks like the letter U (see Figure 13). The smaller the value of s, the sharper the corners get. In practice, s is set to a very small number, for example, 10^{-20}.

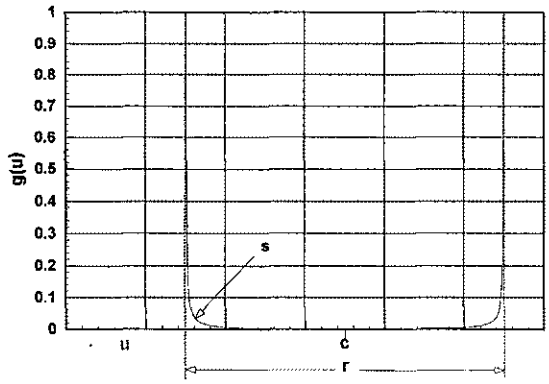

Figure 13. Actuator Constraint Function

To find the minimum of the constraint function, $g(u)$, via Newton-Raphson, one needs to calculate the first and second derivative, and use the negative of the ratio for the step size resulting in

$$step(u) = -\frac{\frac{dg(u)}{du}}{\frac{d^2 g(u)}{du^2}} = \frac{(u-c)(2u-r-2c)(2u+r-2c)}{\left(u-c+\frac{r\sqrt{3}i}{6}\right)\left(u-c-\frac{r\sqrt{3}i}{6}\right)} \tag{28}$$

The function $g(u)$ within the range, r, has two maximums and one minimum, for values of u=-r/2+c, r/2+c, c, respectively (see Figure 14). Thus the Newton-Raphson step size has three corresponding zeros. By inspection of the second derivative we find the maxima are unstable points for Newton-Raphson. Thus, starting in any neighborhood of the maxima, Newton-Raphson will always iterate to the minimum. This would be fine if this were the only minimization required. The cost function for the predictive controller consists of the sum of three parts (17). Since one desires the input constraint to have a negligible effect away from the limits, the Newton-Raphson step size will be dependent solely on the other two parts of the overall cost function. This may cause Newton-

Raphson to step out of the limits even though there is a solution within the limits. In this case the step should be the limit minus some small value ε. The reason for the ε is that if the step brought the input exactly to the limit, no Newton-Raphson iteration would move the input u, since the step size is zero at the maximums. Pushing the step off the maximum allows for convergence to the minimum.

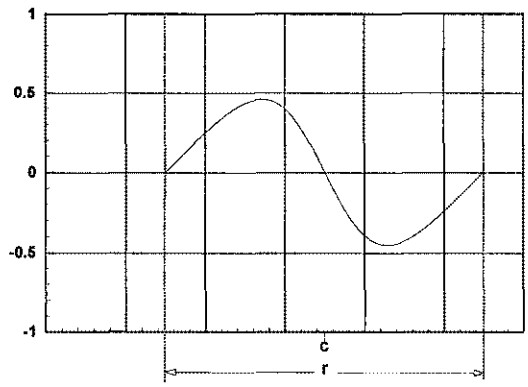

Figure 14. Newton-Raphson Step Size (step(u)).

3.6. Real-Time NGPC Procedure

Sections 3.4 and 3.5 describe the NGPC algorithm for real-time control of most real plants. Presented is a step by step procedure to accomplish real-time control. The approach varies depending on whether the plant is stable or unstable. If the plant is stable, the neural network could be trained off-line to model the plant. This could not be accomplished if the plant is unstable. To train an unstable plant off-line would require the plant to go unstable several times before learning can be accomplished or that a

stabilizing closed-loop system is available. This thesis presents an approach for controlling unstable plants that avoids plant instabilities before learning is accomplished.

The first step to produce stable control is to tune the controller on a simulation of a nominal model of the plant. This will help determine a nominal value of the tuning parameters. There are three tuning parameters N_1, N_2, and λ. The first parameter N_1 is set to the dead time of the plant if it is known otherwise N_1 is set to 1. Setting N_1 to the dead time reduces the computational cost and does not penalize control performance. The running of several simulations are required to determine N_2 and λ. If the plant is minimum phase, start with an N_2=N_1+{order of the plant}. If the plant is nonminimum phase, start with an N_2, that includes the effect of the nonminimum phase components. Run several simulations with several N_2 and several λ and chose the best values. The procedure is outlined in the following:

For Stable Plants:

1. Train the network as described in Section 2.4.4.3 to model that plant.
2. Tune the controller by adjusting N_2 and λ to obtain stable control about the reference signals.
3. If needed, inject input noise on top of the reference signal to obtain richer data to improve the training of the neural network.
4. Take the network off-line and train the network with new data.
5. Place network back on line and do some fine tuning of N_2 and λ.

6. Now place the control on-line with the real plant and repeat this process starting with step 3.

The key to achieving stable control of unstable plants is to initialize the neural network weights with a linear model of the plant. The relationship between the weights and the linear model is described in Section 3.6.1. Since the basic GPC algorithm is robust over modeling errors an accurate linear model is not necessary to produce stable control. All that is required is to have stable control, then the network could learn a better plant model to improve performance.

Unstable Plants

1. Develop a discrete linear model of the plant to be controlled. If the plant is nonlinear, linearize the plant about an operating point.
2. Set the network weights to represent the discrete linear plant as described in Section 3.6.1.
3. Tune the controller by adjusting N_2 and λ to obtain stable control about the reference signals. If control of the plant cannot be obtained for all of the reference signal, then find a subset of the signal that control can be obtained.
4. Inject input noise to obtain richer data to train the neural network.
5. Take the network off-line and train the network with new data as is described in Section 3.6.1.
6. Place network back on-line and fine tune N_2 and λ.

7. Now place the control on-line with the real plant and repeat this process starting with step 4 .

These two procedures are demonstrated in Section 4.

3.6.1. Relation to Linear Difference Equations

NGPC requires a neural network model of the plant to achieve stable control. If the plant is stable, on-line learning can be used to train the network as described in Section 2.4.4.3. Training a network on an unstable plant can cause the plant to go unstable and thus damage the plant. To control an unstable plant without initially training a neural network, an embedded linear model of the plant is necessary. When the network is initialized with an embedded linearized plant, the ability to control the plant becomes a possibility. At this point one could train the network to improve the performance. This will be demonstrated in the inverted pendulum and magnetic levitation plants in the section entitled Case Studies.

There is a close relationship between the time delay networks and linear discrete time filters. The general form of a discrete time filters is described by a linear difference equation with constant coefficients as follows:

$$y(n)+a_1y(n-1)+a_2y(n-2)+\cdots+a_{d_d}y(n-d_d)=b_0u(n)+b_1u(n-1)+\cdots+b_{n_d}u(n-n_d). \quad (29)$$

Solving for $y(n)$ and putting equation (29) into summation form we get:

$$y(n)=\sum_{i=0}^{n_d}\{b_i\,u(n-i)\}-\sum_{i=1}^{d_d}\{a_i\,y(n-i)\} \quad (30)$$

To make a neural network linear, set the network structure to be one hidden layer with one hidden node and no biases, the network's output function, *f*, to be linear with a slope of one, and the output weight to be one, then equations (11) and (12) can be written as:

$$yn(n) = \sum_{i=0}^{n_d} \left\{ w_{1,i+1}\, u(n-i) \right\} + \sum_{i=1}^{d_d} \left\{ w_{1,i+n_d+1}\, y(n-i) \right\}. \tag{31}$$

We see that there is a one-to-one relationship between the weights of the network described by (31) and the coefficients of the difference equation described by (30). This relationship is defined as

$$\begin{bmatrix} w_{1,1} \\ w_{1,2} \\ \vdots \\ w_{1,n_d+1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n_d} \end{bmatrix} \text{ and } \begin{bmatrix} w_{1,n_d+2} \\ w_{1,n_d+2} \\ \vdots \\ w_{1,d_d+n_d+1} \end{bmatrix} = \begin{bmatrix} -a_1 \\ -a_2 \\ \vdots \\ -a_{d_d} \end{bmatrix}.$$

Procedure for Embedding a Linear Model for Control

The network architecture is chosen in the same manner as in Section 2.4.4.3. Choose the first hidden node to have a linear activation function. Set the first hidden node's and output node's bias to zero. Set the weights that corresponded to the connections between the input layer and hidden layer as described above. Set the rest of the weights in the hidden layer as described in Section 2.4.3. Set the weight from the first hidden node to the output node to one, and set the rest of the weights in this layer to zero. The embedded model has been completed. When training this network, one could leave the embedded weights fixed. The rest of the free weights will compensate for any modeling errors. The following is an example of the embedding process.

Example

Consider a network with input nodes consisting of $u(n)$ and two previous inputs (n_d=2), and of three previous outputs (d_d=3), two hidden nodes (h_d=2), and one output node. The difference equation $y(n)+a_1y(n-1)+a_2y(n-2)=b_1u(n-1)$ is the linearized plant model that is to be embedded into the network. Assigning the weights as described above results in a neural network depicted in Figure 15. The weights, $w_{i,j}$'s, are initialized with small random numbers as described in Section 2.4.3. The bias on the first hidden node and output node are set to zero. The other bias is set to a small random number as the $w_{i,j}$'s were.

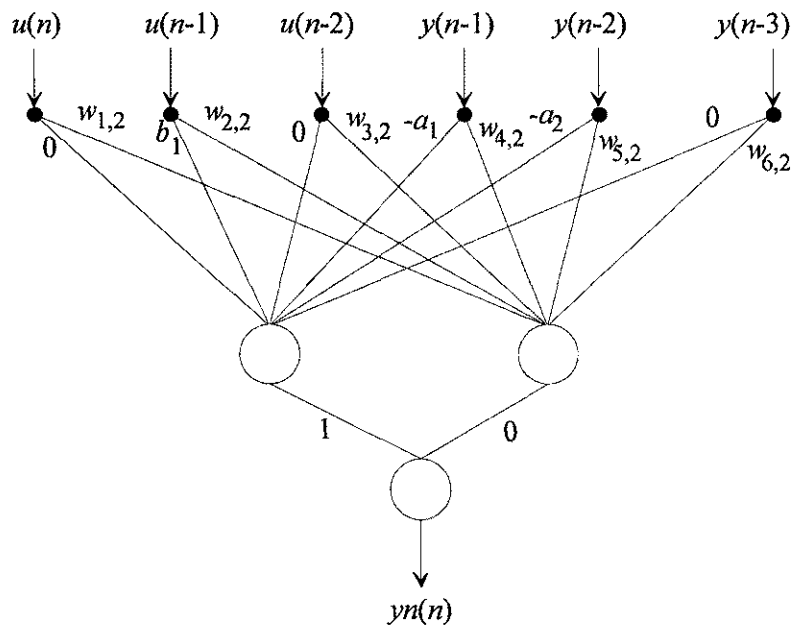

Figure 15. Neural Network with Embedded Linear Model

3.7. Contributions

This chapter develops a Neural Generalized Predictive Control (NGPC) design procedure. The procedure uses a neural network to predict the response of a time-invariant stable or unstable plant. This predictor is used in the minimization of the GPC cost function that is augmented to handle actuator constraints. The cost function is minimized using a Newton-Raphson optimization algorithm. This minimization approach does not appear to have been implemented with GPC or NGPC before. In addition, a technique to control unstable plants before training the neural network predictor is presented. This control design approach will be demonstrated in Chapter 4.

4. Case Studies

4.1. Introduction

When developing a new control algorithm, the first issue that is addressed is whether the controller will deliver the desired performance with robust properties. This has been shown in the work done under GPC and NGPC. The next issue is whether the algorithm can be computed in real-time. NGPC is a very computationally costly algorithm. The main cost comes from the minimization of the cost function. Up to now the algorithms to minimize the cost function have required many iterations for the algorithm to converge. This is the nature of these algorithms. Since Newton-Raphson is a quadratically converging algorithm, the number of iterations are significantly reduced. In the following simulated case studies, only two iterations per sampling instant are necessary for convergence to produce a good control signal.

4.2. Organization of Chapter

Section 4.3 will address programming issues for real-time implementation, and the handling of the input constraint. Section 4.4 will present timing data that supports the use of NGPC for real-time control. Sections 4.5-4.7 will present two case studies showing the performance of the NGPC controller.

4.3. Algorithm comments

Nonlinear optimizations are computationally expensive processes. The use of Newton-Raphson is intended to produce a computationally efficient process. There are many factors that affect the speed of a process, such as algorithm, implementation, rate of convergence, computer, compiler, and problem size. Here we will define the various parameters that dictate the process speed for NGPC.

The algorithm, in conjunction with the complexity of the plant, dictates the rate of convergence. In this thesis the Newton-Raphson optimization has been implemented and for various plants it has been found to converge to a good result within two iterations[4]. All the plants have been modeled with a single hidden layer with three to six hidden nodes, two to four delay nodes on the control input u(n), and three to five delay nodes on the plant input $y(n\text{-}1)$.

For any algorithm, care must be taken to produce efficient code. In this implementation all values that have been calculated in one routine are passed to other routines to avoid recalculation. The most critical portion of the code is the calculation of the Hessian. Two variables, $\partial net_j(n+k)/\partial u(n+h)$ and $\partial yn(n+k)/\partial u(n+h)$, that are calculated in the Jacobian are also used in the calculation of the Hessian. These are passed to the Hessian when it is calculated. Since the Hessian is symmetric, only the upper triangular portion of the matrix needs to be calculated. The use of these two points

[4] A good result is defined here to be less then a 2% change in the control input between interations.

will save a considerable amount of CPU time. For the test case in Section 4.4 the cost of the Hessian was less than the cost of the Jacobian, even though the Jacobian had 5 elements and the Hessian had 25 elements.

Another component of process speed is the computer's performance. To ensure optimal speed the computer, compiler, and floating point data types used were all 32 bit. To be specific a Pentium 90 MHz 16 MB Micronics motherboard, with Microsoft Visual C++ version 2.0 using full optimization, Pentium specific compiling and in-lining whenever possible was used. In-lining is instrumental for speed and readability of the module code.

The above factors were used in the following section to demonstrate the speed of the algorithm.

4.4. Timing Specifications

Section 4.1 outlines various factors that determine the speed of a process. To present timing data a nominal plant model and various cost horizons N_2 were selected. The timing data was collected with a network with two inputs, four delays on the first input, five delays on the second input, and six hidden nodes. This was chosen to generate an upper lower bound on the speed of the process with respect to the case studies. The following case studies use networks equal to or smaller in size than this nominal network,

therefore, the case studies could execute faster than what is presented here. The variable N_2 was varied from 1 to 10 to generate the timing data in Figure 16. The data is shown in terms of servo rate plotted on a semi-log plot to shown the exponential decrease in servo rate as N_2 increases.

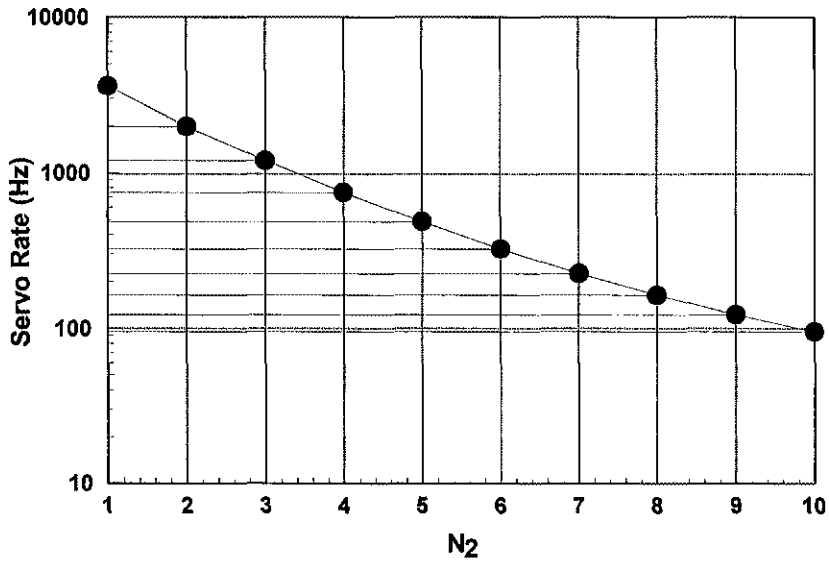

Figure 16. Timing Data for NGPC

To measure the efficiency of the NGPC algorithm the computational cost is broken down into five calculations, the Jacobian, Hessian, plant prediction, LU decomposition, and miscellaneous over head. The case for N_2=5 is presented in Table 1 as a percentage of computational cost.

Routine	Percent Time
$\partial J / \partial U$	37.48
$\partial J^2 / \partial^2 U$	32.72
Prediction	21.89
Solution	6.19
Misc.	1.72

Table 1. Percentage of Time for Key Routines

Since the cost of the Hessian would not be included in a first order gradient technique, this time can be eliminated when comparing this Newton-Raphson implementation to the gradient technique found in [28]. The calculation of the Hessian takes 32.72% of the CPU time. Without the Hessian calculations and the LU decomposition, the percent CPU time used for an iteration is 61.09%. Using this percent time, the gradient algorithm would be able to calculate 1.64 iterations for the same CPU time. Since the gradient algorithm in [28] takes 10 to 20 iterations, the Newton-Raphson algorithm runs 6.1 to 12.2 times faster.

4.5. Mass-Spring-Damper with stiffening spring

Plant Model

The first of the two case studies is the control of a mass-spring-damper with a stiffening spring represented by the Duffing's equation

$$\ddot{y}(t) + \dot{y}(t) + y(t) + y^3(t) = u(t). \tag{32}$$

The plant was simulated using a fourth order Runga-Kutta integration routine [22] with an integration step size of 0.2 seconds. The output of the integration was sample at a rate of 0.2 seconds. Figure 17 and 18 shows the response of the plant due to a series of pulses with increasing amplitude. The nonlinearity of this plant can be observed by noting that a linear increase in input amplitude does not result in a proportional increase in the output

and a frequency shift is also produced. This plant will be controlled to track a reference signal similar to Figure 17 but with a maximum amplitude of one.

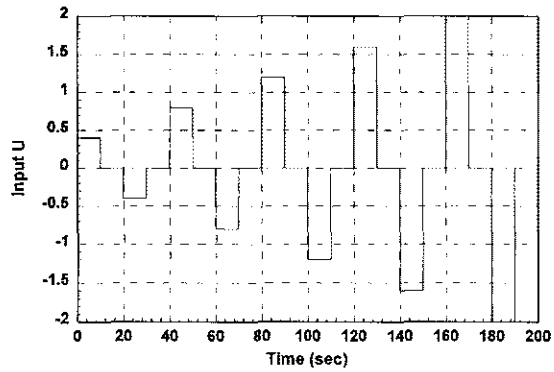

Figure 17. Pulse Train Input

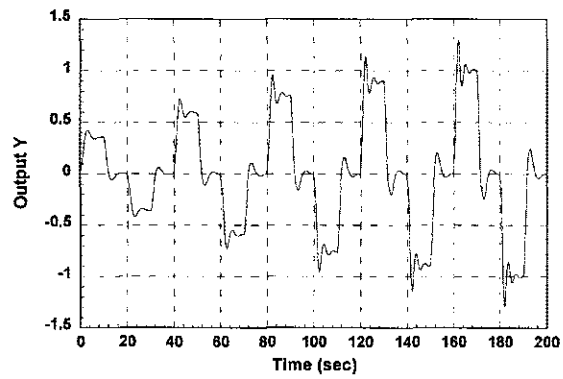

Figure 18. Duffing's Equation Response to the Pulse Train

Network Structure

The network structure is detailed in Table 2. The number of delay nodes are chosen as described in Section 2.4.4. The set of parameters in Table 2 gives acceptable results.

Input Delays	2
Feedback Delays	3
Hidden Nodes	5

Table 2. Network Structure for Duffing's Equation

Weight Initialization and Network Learning

The neural network was trained to model the plant using the same pulse train. The network architecture contained a single hidden layer with five hidden nodes and a single output node. The input layer was composed of two inputs, one externally fed with 2 time delayed nodes and the other from the output of the plant with 3 time delayed nodes. The hidden layer nodes used the hyperbolic tangent as an activation function and the output was scaled linearly. Normalized Root Mean Squared error (NRMS) and Max error measures were used in the training of the network. Figures 19 and 20 shows both the NRMS and Max errors for 1,000 cycles of network training, respectively. The NRMS error is well below 10^{-2}.

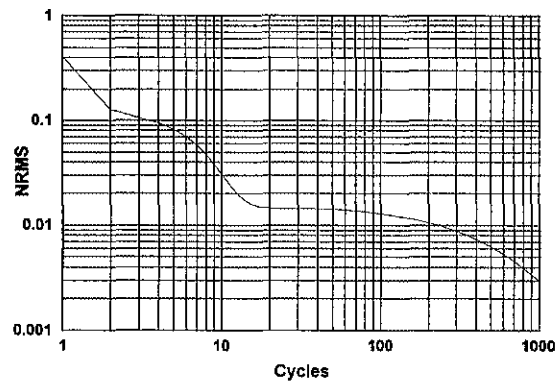

Figure 19. NRMS for Duffing's Equation Training

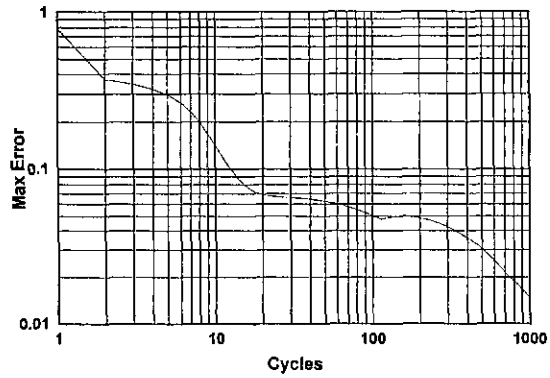

Figure 20. Max Error for Duffing's Equation Training

Figure 21 and 22 compares the response of the plant and the trained neural network and their corresponding error, respectively.

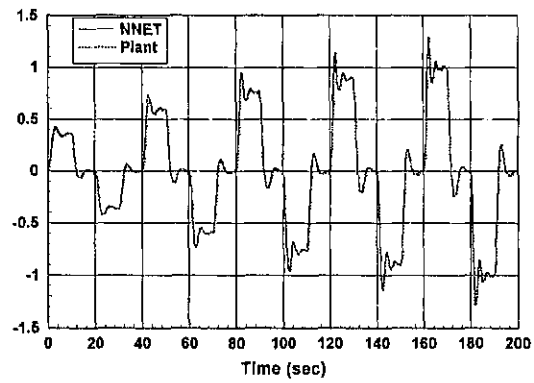

Figure 21. Plant and Network Response for Duffing's Equation

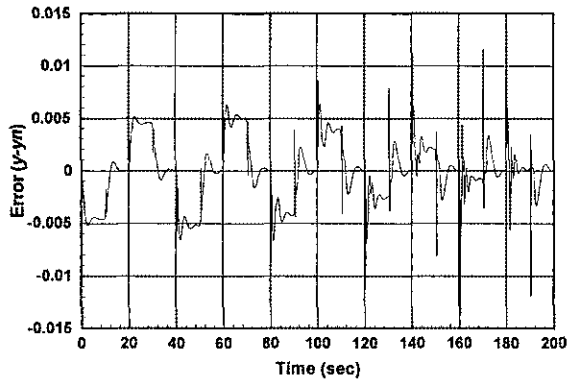

Figure 22. Error Between Plant and Network for Duffing's Equation

Plant Control

The nonlinear model of the plant was placed into the NGPC control-loop where the reference model is the pulse train. The system was tuned by varying the horizon, N_2, and λ to produce a desirable response. The final tuning resulted in a horizon of five and a value of 0.0001 for λ. Figure 23 and shows the controlled response of the plant tracking the pulse train and the error between the plant and the reference model.

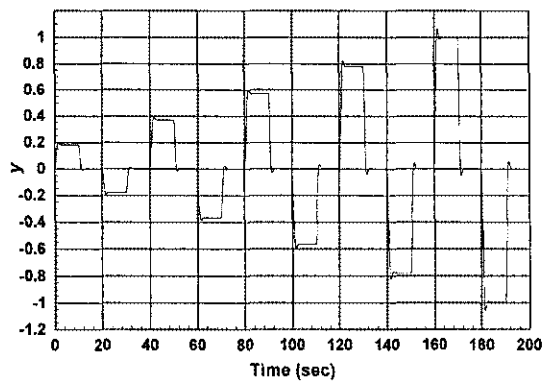

Figure 23. Controlled Response of the Plant for Duffing's Equation

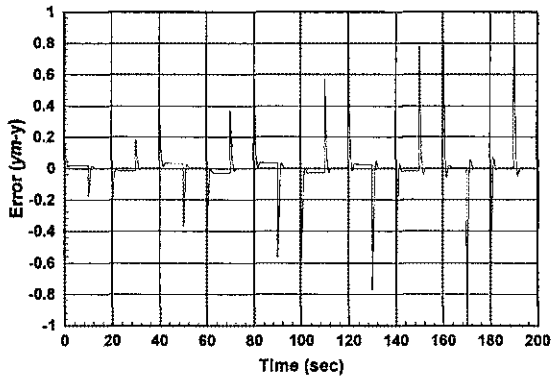

Figure 24. Tracking Error for Duffing's Equation

To compare these results to results where the model is linear, equation (32) was linearized about zero. Discretizing with a sample time of 0.2 seconds with the step invariant transform produces

$$y(n) - 1.783y(n-1) + 0.8187y(n-2) = 0.01867u(n-1) + 0.01746u(n-2). \qquad (33)$$

Equation (33) replaced the neural network as the plant estimator. Figure 25 shows the controlled response of the plant with the linear plant model. Figure 26 shows the tracking error. Comparing results shown in Figure 24 and 26 we see that the linear model controller had significantly higher steady state error, thus demonstrating the benefit in using a neural network for the plant estimator.

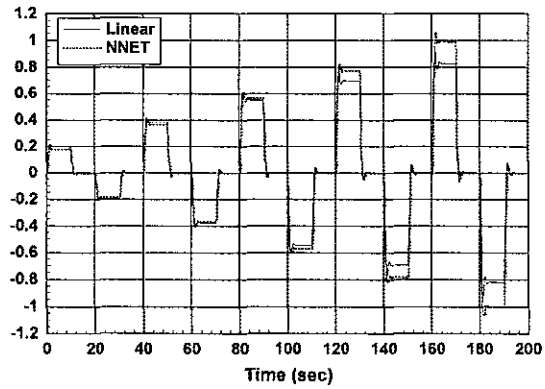

Figure 25. Controlled Response with Linear Model

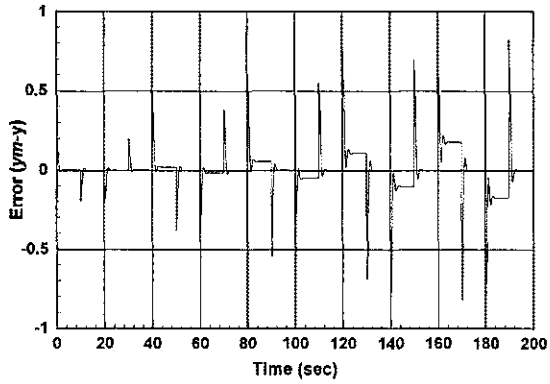

Figure 26. Tracking Error with Linear Model

4.6. Magnetic Levitation

Plant Model

The magnetic levitation plant consists of an electric magnetic and a steel ball. The steel ball is levitated under the electro-magnet. The electro-magnet is driven with a current input $i(t)$. The steel ball is measured in millimeters starting at the bottom of the electro-magnet, see Figure 27.

A nominal nonlinear plant model for this single degree of freedom magnetic levitation system was described in [29]. The nonlinear equation of motion is

$$\ddot{x}(t) = g - \frac{i^2(t)}{c(x(t))m} \tag{34}$$

where

$c(x(t)) = \frac{(a_0 + a_1 x(t))^2}{c_2}$ is a polynomial approximation of force-current characteristics

m is the mass of the steel ball,

$i(t)$ is the input current, and

g is the gravitational constant.

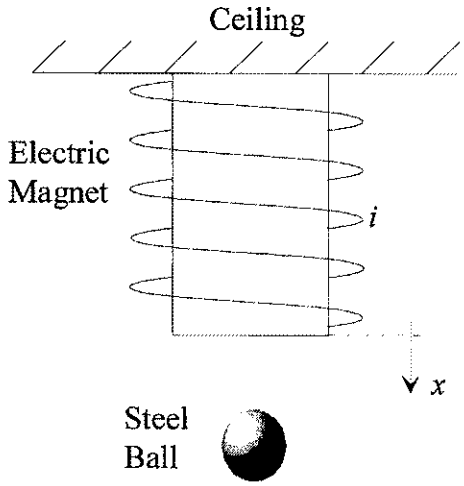

Figure 27. Magnetic Levitation Plant

Network Structure

The network structure is detailed in Table 3. The number of delay nodes were chosen as described in Section 2.4.4. The set of parameters in Table 5 gives acceptable results.

Weight Initialization and Plant Modeling

For initial stable control and for off line tuning an initial linear model of the plant was needed to initialize the NGPC neural model. This was accomplished by linearizing and discretizing the nominal model of the plant. Linearizing the plant model about an operating point, x_0, resulted in

$$\ddot{x}(t) = g - \frac{m_0 i_0 i(t) + i_b}{c(x_0) m} \tag{35}$$

where

x_0 is the point which the plant is linearized about

i_0 is the equilibrium current at x_0

m_0 is the slope of $i^2(t)$ at x_0 which is 2 for all x_0

i_b is the y intercept of the linear model for $i^2(t) = i_0^2$ (i.e. solution to $i_0^2 = m_0 i_0^2 + i_b$)

Rearranging (35) to separate the gravitational effects and the current input $i(t)$ we got

$$\ddot{x}(t) = \frac{g\, c(x_0) m - i_b}{c(x_0) m} u_s(t) - \frac{m_0 i_0}{c(x_0) m} i(t), \tag{36}$$

Input Delays	4
Feedback Delays	3
Hidden Nodes	4

Table 3. Network Structure for the Magnetic Levitation Plant

where

$u_S(t)$ is the unit step function.

Define the input to be $u(t)=\frac{g\,c(x_0)m-i_b}{c(x_0)m}u_s(t)-\frac{m_0\,i_0}{c(x_0)m}i(t)$ and substituting this into Equation (36) we have $\ddot{x}(t)=u(t)$. Taking the Laplace transform and representing the linearized plant in transfer function form we have $G(s)=\frac{X(s)}{U(s)}=\frac{1}{s^2}$. Discretizing using step invariance technique we have $G(z)=\frac{T^2(z+1)}{2(z-1)^2}$. Applying the inverse z transform and substituting a discretized $u(t)$, $u(n)=\frac{g\,c(x_0)m-i_b}{c(x_0)m}u_s(n)-\frac{m_0\,i_0}{c(x_0)m}i(n)$ we get the difference equation

$$x(n)=2x(n-1)-x(n-2)+\frac{T^2}{2}\left(\frac{g\,c(x_0)m-i_b}{c(x_0)m}\left[u_s(n-1)+u_s(n-2)\right]-\frac{m_0\,i_0}{c(x_0)m}\left[i(n-1)+i(n-2)\right]\right) \quad (37)$$

for the linearized discretized plant. The constant values for i_0 and i_b were solved for as follows. When the plant is in equilibrium between the gravitational force and force due to the input current then $i(t)=i_0$ and the acceleration $\ddot{x}(t)=0$. From Equation (36) we have

$$g=\frac{m_0\,i_0\,i(t)-i_b}{c(x_0)m} \quad (38)$$

and from the linear model of $i^2(t)$

$$i_b=i_0^2(1-m_0). \quad (39)$$

Substitution i_b in (38) and solving for i_0 we got

$$i_0 = \sqrt{g\,c(x_0)\,m}\,.$$

Substituting m_0=2 and i_0 into (39) we got

$$i_b = -g\,c(x_0)\,m$$

Substituting i_b into the difference Equation (37) we got

$$x(n) = 2x(n-1) - x(n-2) + T^2\left(g\left[u_s(n-1) + u_s(n-2)\right] - \frac{m_0\, i_0}{2c(x_0)\,m}\left[i(n-1) + i(n-2)\right]\right) \quad (40)$$

Since the difference Equation (40) was used to embed a linear model into the neural network special consideration needed to be paid to how to handle the gravitational term. From this equation we see that the effect of gravity starts at n=1 and doubles at n=2. After that, the effects of gravity is constant. This constant was represented by the bias term on the first node of the hidden layer. An initial condition error was introduced by assuming that gravity is present in full force at n=0. This error is small and dies out rapidly. NGPC is robust enough that this error does not significantly affect the overall system response.

Table 4 shows the parameters from [29], and the solution of the above equations for i_0 and i_b. Substituting the parameters from Table 4 into (40) and neglecting the two step functions we got

$x(n)$=2$x(n$-1)-$x(n$-2)-0.000074770[$i(n$-1)+$i(n$-2)]+0.0000392402

as the linearized difference equation of motion for the magnetic levitation plant. The first hidden node weights was set to represent this difference equation as described in Section 3.6.1. The other weights were set to very small random numbers as described in Section 2.4.4. The bias on the first hidden node was set to the gravitational effects constant.

Plant Control

Initial control and tuning was accomplished with the embedded model. The system was tuned by varying the horizon, N_2, and λ to produce a desirable response. The final tuning resulted in a horizon of five and a value of 0.001 for λ. The response for a reference model being a filtered pulse train, where the filter is third order repeated pole at 10 rad/sec, is shown in Figure 28. The control input, u, is shown in Figure 29. The response shows that the embedded model did stabilize the plant but did not result in a good performance.

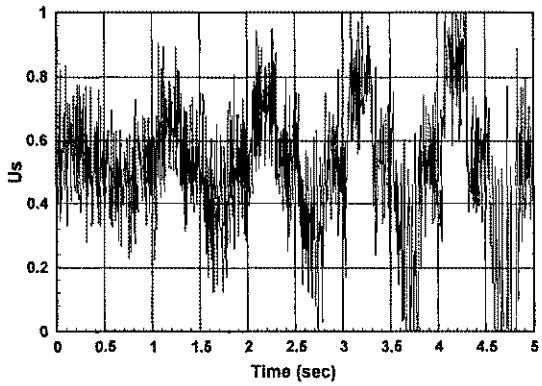

Figure 28. Control Input for the Magnetic Levitation Plant

Parameter	Value
T	0.002 sec
g	9.81 m/sec^2
m	0.0682 kg
x_0	0.01 m
a_0	1.2084
a_1	460.16
c_2	82.0
$c(x_0)$	0.41166
i_b	-0.27542 amps
i_0	0.5248 amps

Table 4. Parameters for the Magnetic Levitation Plant

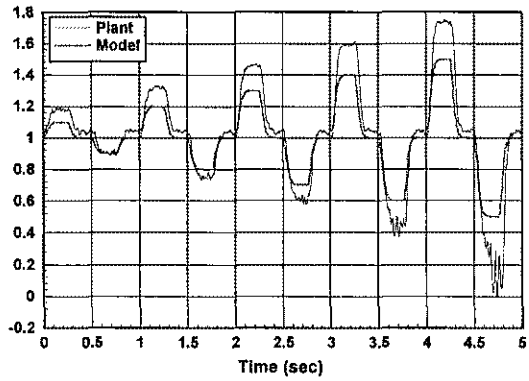

Figure 29. Untrained Response of the Magnetic Levitation Plant

Using the data from the above response the network was trained and then controlled. Under control we found that we achieved good tracking with much less control action then before training. The control input and plant response is shown in Figures 30 and 31 respectively.

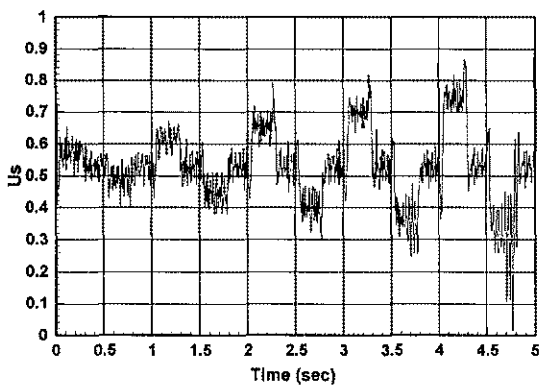

Figure 30. Control Input after Training for the Magnetic Levitation Plant

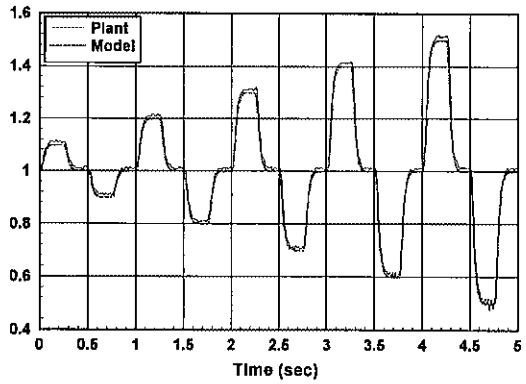

Figure 31. Response after Training of the Magnetic Levitation Plant

4.7. Conclusions

This chapter presents a timing analysis of the NGPC algorithm and demonstrates its applicability in two computer case studies. The simulation studies show that NGPC can control non-linear unstable systems. It is also shown in one of the case studies that a neural network predictor results in better performance than a linear predictor model. This is expected since the plant was nonlinear. The typical prediction horizon used was five, thus enabling 500 Hz servo rate control based on the timing data.

5. Conclusions

5.1. Conclusions

The main contributions of this thesis are the development of a computationally efficient Neural Generalized Predictive Controller, the augmented cost function that handles actuator saturation, and the real-time procedure to control unstable plants with an untrained neural network. NGPC is made computationally efficient by utilizing the Newton-Raphson optimization algorithm to minimize the GPC cost function..

Algorithm timing analysis and two case studies were presented to demonstrate the speed and capability of NGPC. Timing the speed of the algorithm shows that a typical servo rate of 500 Hz. is attainable with a Pentium 90 MHz PC. The simulation results of controlling Duffing's equation shows the improved control performance of NGPC over GPC with a linear model. Simulation results of two unstable plants shown demonstrated the technique of embedding a linear model into the neural network for initial stable control.

5.2. Future research

This thesis serves as a foundation for the implementation of the original GPC cost function for SISO plants using a neural network as the plant model. Extensions to MIMO and improvements of the cost function is of interest as future research. Other issues such as sensitivity to sensor noise and on-line adaptation are topics for future research.

References

[1] Charles Anderson, "Learning to Control an Inverted Pendulum Using Neural Networks," *Proceedings of the American Control Conference*, Atlanta, Georgia, June 15-17, 1988.

[2] W. T. Miller III, R. S. Sutton, and P. J. Werbos, "Neural Networks for Control" *MIT Press* 1990.

[3] D. A. White and D. A. Sofge, "Handbook of Intelligent Control" *Van Nostrand Reinhold* 1992.

[4] K.S. Narendra and K. Parthasarthy, "Identification and Control of Dynamical Systems using Neural Networks", *IEEE Transactions on Neural Networks*, March 1990.

[5] D. E. Rumelhart and J. L. McClelland, "Parallel Distributed Processing", *MIT Press*, 1986.

[6] T. J. Sejnowski and C. R. Rosenberg, "NETalk: A Parallel Network That Learns to Read Aloud" JHU/EECS-86/01, School of Electrical Engineering and Computer Science, John Hopkins University, 1986.

[7] S. Grossberg and M. Kuperstein, "Neural Dynamics of Adaptive Sensory-Motor Control." *Elsevier Science Publishers*, 1986.

[8] K. Hornik, M. Stinchcombe, and H. White, "Multilayer Feedforward Network are Universal Approximaters," Neural Networks Vol. 2, pp. 359-366.

[9] Jeong Jun Song and Sunwon Park, "Neural Model-Predictive Control for Nonlinear Chemical Processes," *Journal of Chemical Engineering of Japan*, V26, N4, pp. 347-354, 1993.

[10] G. A. Montague, M. J. Willis, M. T. Tham and A. J. Morris, "Artificial Neural Network Based Control," *International Conference on Control 1991*, Vol. 1 pp. 266-271.

[11] D. C. Psichogios and L. H. Ungar, "Nonlinear Internal Model Control and Model Predictive Control using Neural Networks," *5th IEEE International Symposium on Intelligent Control 1990*, pp.1082-1087.

[12] Y. Takahashi, "Adaptive Predictive Control of Nonlinear Time-Varying System using Neural Network," *1993 IEEE International Conference on Neural Networks*, Vol. 3, pp. 1464-1468.

[13] G. Davis, "Numerical Methods in Engineering & Science," *Allen & Unwin*, 1986.

[14] G. Ake Bjorck, "Numerical Methods," *Prentice-Hall*, 1974.

[15] J.J. Hopfield and D. W. Tank, "Neural Computation of Decisions in Optimization Problems," *Biological Cybernetics*, 1985, Vol 52, pp. 141-152.

[16] T. Kohonen, "Self-organization and Associative Memory," Springer Series in *Information Sciences* 8, Springer-Verlag, 1984.

[17] A. J. Owens and D. L. Filkin, "Efficient Training of the Back Propagation Network by Solving a System of Stiff Ordinary Differential Equations," *Proceedings International Joint Conference on Neural Networks*, June 1989, Washington, DC, Vol. 2, pp. 381-386.

[18] T. P. Vogl, J.K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alkon, "Accelerating the Convergence of the Back-Propagation Method," *Biological Cybernetics*, 1988, Vol 59, pp. 257-263.

[19] A. G. Barto and M. I. Jordan, "Gradient Following without Back-Propagation in Layered Networks," *IEEE 1st International Conference on Neural Networks*, 21-24 June 1987, San Diego, CA, pp. 629-639.

[20] D. W. Clarke, C. Mohtadi and P. S. Tuffs, "Generalized Predictive Control - Part I. The basic Algorithm," *Automatica*, Vol. 23, No 2, pp. 137-148, 1987.

[21] D. W. Clarke, C. Mohtadi and P. S. Tuffs, "Generalized Predictive Control - Part II. The basic Algorithm," *Automatica*, Vol. 23, No 2, pp. 149-163, 1987.

[22] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, "Numerical Recipes in C: The Art of Scientific Computing," *Cambridge University Press* 1988.

[23] J. J. Shynk, "Adaptive IIR Filtering," IEEE ASSP Magazine, pp. 4-21, 1989.

[24] K. J. Hunt and D. Sbarbaro, "Adaptive Filtering and Neural Networks for Realization of Internal Model Control," *IEEE Intelligent Systems Engineering*, pp. 67-76, Summer 1993.

[25] D. W. Clarke, "Advances in model-based predictive control," in *Advances in Model-Based Predictive Control*, ed. by D. W. Clarke, Oxford Univ. Press, 1994.

[26] Jeong Jun Song and Sunwon Park, "Neural Model-Predictive Control for Nonlinear Chemical Processes," *Journal of Chemical Engineering of Japan*, 1993, V26, N4, pp. 347-354.

[27] D. C. Psichogios and L. H. Ungar, "Nonlinear Internal Model Control and Model Predictive Control using Neural Networks," *5th IEEE International Symposium on Intelligent Control 1990*, pp.1082-1087.

[28] H. Koivisto, P. Kimpimaki, H. Koivo, "Neural Predictive Control - A Case Study," *Proceedings of the 1991 IEEE International Symposium on Intelligent Control*, 13-15 August 1991, Arlington, Virginia, U.S.A, pp. 405-410.

[29] S. Mercurio, I. Rusnak, W.S. Gray, and M. Kam, "Controlling a Magnetic Levitation System via Feedback Linearization," *1992 Conference on Information Sciences and System*, Vol. 2, Princeton, New Jersey, pp. 1030-1035.

[30] S. Haykin, "Neural Networks: A Comprehensive Foundation," *Macmillan College Publishing Company, Inc.* 1994.

[31] K. Astrom and B. Wittenmark, "Adaptive Control," *Addison-Wesley*, 1989.

[32] T. T. Ho, J. T. Bialasiewicz and E. T. Wall, "On Stochastic Minimum Variance Adaptive Control," *Proceedings of the 7th International Conference on Systems Engineering*, Las Vegas, Nevada, July 1990.

[33] D. Nguyen and B. Widrow, "Neural Networks for Self-Learning Control Systems," *IEEE Control Systems Magazine*, April 1990, pp. 18-23.

[34] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike Elements that can Solve Difficult Learning Control Problems," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 13, pp. 835-846, 1983.

[35] H. Gomi and M. Kawato, "Learning Control for a Closed Loop System using Feedback-Error-Learning," *Proceedings of the 20th Conference on Decision and Control*, December 1990, Honolulu, Hawaii, pp. 3289-3294.

[36] M. Kawato, K. Furukawa, and R Suzuki, "A Hierarchical Neural-Network Model for Control and Learning of Voluntary movement," *Biological Cybernetics*, Vol 57, pp. 169-185.

[37] M. Riedmiller, "Aspects of Learning Neural Control," *1994 IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 2, pp. 1458-1463.

[38] D. Psaltis, A. Sideris, and A. Yamamura, "A Multilayered Neural Network Controller," *IEEE Control System Magazine*, Vol. 8, pp. 17-21.

[39] M. Minsky and S. Papert, "Perceptrons," *The MIT Press* 1969.

[40] M. Jordan and R. Jacobs, "Learning to Control an Unstable System with Forward Modeling," *Advances in Neural Information Processing Systems 2*, San Mateo, CA, 1990.

[41] J. Saint-Donat, N. Bhat, and T. McAvoy, "Neural Net Based Model Predictive Control," *International Journal of Control*, 1991, Vol. 54, No. 6, pp. 1453-1468.

[42] Hassoun, Mohamad H. "Fundamentals of Artificial Neural Networks," 1995, *Mass. Institute of Tech. Press*, page 99.