NASA Technical Memorandum 110244

Neural Generalized Predictive Control: A Newton-Raphson Implementation

Donald Soloway and Pamela J. Haley
Langley Research Center, Hampton, Virginia

February 1997

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

NEURAL GENERALIZED PREDICTIVE CONTROL
A Newton-Raphson Implementation

Donald Soloway
Pamela J. Haley
NASA Langley Research Center
Hampton, VA. 23681
don@ptolemy.arc.nasa.gov
p.j.haley@larc.nasa.gov

Abstract

An efficient implementation of Generalized Predictive Control using a multi-layer feedforward neural network as the plant's nonlinear model is presented. In using Newton-Raphson as the optimization algorithm, the number of iterations needed for convergence is significantly reduced from other techniques. The main cost of the Newton-Raphson algorithm is in the calculation of the Hessian, but even with this overhead the low iteration numbers make Newton-Raphson faster than other techniques and a viable algorithm for real-time control. This paper presents a detailed derivation of the Neural Generalized Predictive Control algorithm with Newton-Raphson as the minimization algorithm. Simulation results show convergence to a good solution within two iterations and timing data show that real-time control is possible. Comments about the algorithm's implementation are also included.

Introduction

Generalized Predictive Control (GPC), introduced by Clarke and his coworkers in 1987, belongs to a class of digital control methods called Model-Based Predictive Control (MBPC) [4,5,14]. MBPC techniques have been analyzed and implemented successfully in process control industries since the end of the 1970's and continue to be used because they can systematically take into account real plant constraints in real-time. GPC is known to control non-minimum phase plants, open-loop unstable plants and plants with variable or unknown dead time. It is also robust with respect to modeling errors, over and under parameterization, and sensor noise [4]. GPC had been originally developed with linear plant predictor models which leads to a formulation that can be solved analytically. If a nonlinear model is used a nonlinear optimization algorithm is necessary. This affects the computational efficiency and performance by which the control inputs are determined. For nonlinear plants, the ability of the GPC to make accurate predictions can be enhanced if a neural network is used to learn the dynamics of the plant instead of standard nonlinear modeling techniques. The selection of the minimization algorithm affects the

computational efficiency of the algorithm. In using Newton-Raphson as the optimization algorithm, the number of iterations to convergence is significantly reduced from other techniques. The main cost of the Newton-Raphson algorithm is in the calculation of the Hessian, but even with this overhead the low iteration numbers make Newton-Raphson a faster algorithm for real-time control.

The Neural Generalized Predictive Control (NGPC) system can be seen in Figure 1. It 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 net block.

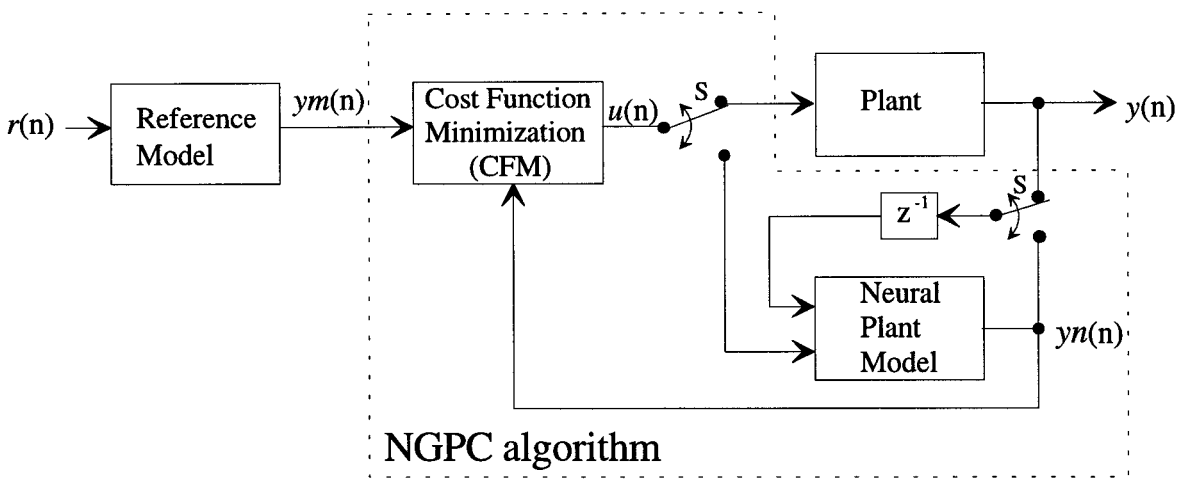

Figure 1. Block diagram of the NGPC system and algorithm

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 a specified cost function. Between samples, the switch is set to the plant's model where the CFM algorithm uses this model to calculate the next control input, u(n+1), from predictions of the response from the plant's model. Once the cost function is minimized, this input is passed to the plant. This algorithm is outlined below.

The NGPC algorithm has the following important steps.

1) Generate a reference trajectory. If the future trajectory of $ym(n)$ is unknown, keep $ym(n)$ constant for the future trajectory.
2) Start with the previous calculated control input vector, and predict the performance of the plant using the model.

3) Calculate a new control input that minimizes the cost function,
4) Repeat steps 2 and 3 until desired minimization is achieved,
5) Send the first control input, to the plant,
6) Repeat entire process for each time step.

The computational performance of a GPC implementation is largely based on the minimization algorithm chosen for the CFM block. There are several minimization algorithms that have been implemented in GPC such as Non-gradient [11], Simplex [13], and Successive Quadratic Programming [10,12]. The selection of a minimization method can be based on several criteria such as; number of iterations to a solution, computational costs and accuracy of the solution. In general these approaches are iteration intensive thus making real-time control difficult. Very few papers address real-time implementation or the papers use plants that have a large time constant [16,17]. 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.

The quality of the plant's model affects the accuracy of a prediction. A reasonable model of the plant is required to implement GPC. With a linear plant there are tools and techniques available to make modeling easier, but when the plant is nonlinear this task is more difficult. Currently there are two techniques used to model nonlinear plants. One 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. The second 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. Models using neural networks have been shown to have the capability to capture nonlinear dynamics [3]. For nonlinear plants, the ability of the GPC to make accurate predictions can be enhanced if a neural network is used to learn the dynamics of the plant instead of standard modeling techniques. Improved predictions affect rise time, over-shoot, and the energy content of the control signal.

This paper is divided into nine sections with the introduction being the first section. The second section begins by describing the cost function that the GPC algorithm uses to compute the control input. This cost function is minimized by Newton-Rhapson to obtain a solution. Newton-Rhapson, the cost function minimization algorithm, is derived in section three. The NGPC algorithm uses a trained neural network as the plant's model. The network equations are found in section four and prediction using a neural network is described in section five. The sixth section derives the derivative equations of the network needed for the CFM. The sixth section shows simulation results followed by section seven, timing specifications. The eighth section gives further optimizations that were taken advantage of during implementation that significantly improved the computational overhead. Finally, the last section, section nine concludes the paper.

The Cost Function

As mentioned earlier, the NGPC algorithm is based on minimizing a cost function over a finite prediction horizon. The cost function of interest to this application is

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

where

N_1 is the minimum costing horizon,
N_2 is the maximum costing horizon,
N_u is the control horizon,
ym is a reference trajectory,
yn is the predicted output of the neural network,
λ is the control input weighting factor,
$\Delta u(n+j)$ is the change in *u* and is defined as $u(n+j)-u(n+j-1)$,
s is the sharpness of the corners of the constraint function,
r is the range of the constraint, and
b is an offset to the range.

This cost function minimizes not only the mean squared error between the reference signal and the plant's model, but also the weighted squared rate of change of the control input with it's constraints.

When this cost function is minimized, a control input that meets the constraints is generated that allows the plant to track the reference trajectory within some tolerance.

There are four tuning parameters in the cost function, N_1, N_2, N_u, and λ. The predictions of the plant will run from N_1 to N_2 future time steps. The bound on the control horizon is N_u. The only constraint on the values of N_u and N_1 is that these bounds must be less than or equal to N_2. The second summation contains a weighting factor, λ, that is introduced to control the balance between the first two summations. The weighting factor acts as a damper on the predicted $u(n+1)$. The third summation of *J* defines constraints placed on the control input. The parameters *s*, *r*, and *b* characterize the sharpness, range, and offset of the input constraint function respectively. The sharpness, *s*, controls the shape of the constraint function. When plotted, the constraint function looks like the letter U. 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}.

The Cost Function Minimization Algorithm

The objective of the CFM algorithm is to minimize J in (1) with respect to $[u(n+1), u(n+2), \ldots, u(n+N_u)]^T$, denoted $\mathbf{U}$. This is accomplished by setting the Jacobian of (1) to zero and solving for $\mathbf{U}$. With Newton-Rhapson used as the CFM algorithm, J is minimized iteratively to determine the best $\mathbf{U}$. 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

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

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

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

where the Jacobian is denoted as

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

and the Hessian as

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

Solving Equation (2) 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 [7] to solve for the control input vector $\mathbf{U}(k+1)$. This is accomplished by rewriting Equation (2) in the form of a system of linear equations, $\mathbf{Ax} = \mathbf{b}$. This results in

$$\frac{\partial^2 \mathbf{J}}{\partial \mathbf{U}^2}(k)(\mathbf{U}(k+1) - \mathbf{U}(k)) = -\frac{\partial \mathbf{J}}{\partial \mathbf{U}}(k), \tag{3}$$

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 (2) can be solved with two routines supplied in [7], the LU decomposition routine, ludcmp, and the system of linear equations solver, lubksb.

After **x** is calculated, **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 **U**(k+1) is less than some ε. When solving for **x**, calculation of each element of the Jacobian and Hessian is needed for each Newton Rhapson iteration. The hth element of the Jacobian is

$$\frac{\partial J}{\partial u(n+h)}=-2\sum_{j=\mathrm{N}_1}^{\mathrm{N}_2}\left[ym(n+j)-yn(n+j)\right]\frac{\partial yn(n+j)}{\partial u(n+h)}+2\sum_{j=1}^{\mathrm{N}_u}\lambda(j)\left[\Delta u(n+j)\right]\frac{\partial \Delta u(n+j)}{\partial u(n+h)}$$
$$+\sum_{j=1}^{\mathrm{N}_u}\delta(h,j)\left[\frac{-s}{\left(u(n+j)+\frac{r}{2}-b\right)^2}+\frac{s}{\left(\frac{r}{2}+b-u(n+j)\right)^2}\right]\quad ,h=1,\ldots,\mathrm{N}_u.$$

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[1],

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

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

$$\frac{\partial^2 J}{\partial u(n+m)\partial u(n+h)}=2\sum_{j=\mathrm{N}_1}^{\mathrm{N}_2}\left\{\frac{\partial yn(n+j)}{\partial u(n+m)}\frac{\partial yn(n+j)}{\partial u(n+h)}-\frac{\partial^2 yn(n+j)}{\partial u(n+m)\partial u(n+h)}\left[ym(n+j)-yn(n+j)\right]\right\}$$
$$+2\sum_{j=1}^{\mathrm{N}_u}\lambda(j)\left\{\frac{\partial \Delta u(n+j)}{\partial u(n+m)}\frac{\partial \Delta u(n+j)}{\partial u(n+h)}+\Delta u(n+j)\frac{\partial^2 \Delta u(n+j)}{\partial u(n+m)\partial u(n+h)}\right\}$$
$$+\sum_{j=1}^{\mathrm{N}_u}\delta(h,j)\delta(m,j)\left[\frac{2s}{\left(u(n+j)+\frac{r}{2}-b\right)^3}+\frac{2s}{\left(\frac{r}{2}+b-u(n+j)\right)^3}\right]\quad \begin{matrix},h=1,\ldots,\mathrm{N}_u\\ ,m=1,\ldots,\mathrm{N}_u\end{matrix}.$$

---

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

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

The last component needed to evaluate **U**(k+1) is the calculation of the output of the plant, $yn(n+j)$, and it's derivatives. The next three sections define the equations of a multi-layer feedforward neural network, describe the prediction process using a neural network, and define the derivative equations of the neural network.

Neural Network Architecture

In NGPC the model of the plant is a neural network. The initial training of the neural network is typically done off-line before control is attempted. The block configuration for training a neural network to model the plant is shown in Figure 2. The network and the plant receive the same input, u(n). The network has an additional input that either comes from the output of the plant, $y(n)$, or the network, $yn(n)$. The one that is selected depends on the plant and the application [9]. This input assists the network with capturing the plant's dynamics and stabilization of unstable systems. To train the network, its weights are adjusted such that a set of inputs produces the desired set of outputs. An error is formed between the responses of the network, yn(n), and the plant, $y(n)$. This error is then used to update the weights of the network through gradient descent learning [2]. This process is repeated until the error is reduced to an acceptable level.

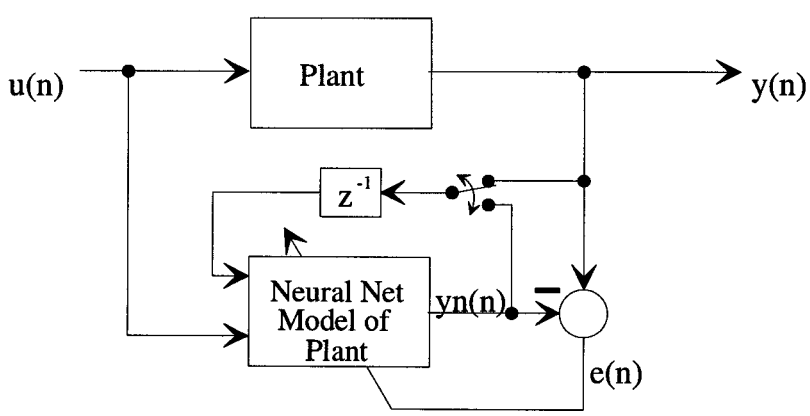

Figure 2. Block diagram of off-line neural network training

Since a neural network will be used to model the plant, the configuration of the network architecture should be considered [3]. A neural network can be configured for either

input/output or state space modeling. This implementation of NGPC adopts input/output models since state space modeling requires measurements of the plant's states that are not always available. The diagram below, Figure 3, depicts a multi-layer feedforward neural network with a time delayed structure. For this example, the inputs to this network consists of external inputs, u(n) and y(n-1), and their corresponding delay nodes, u(n-1), ..., u(n-n_d), and y(n-2), ..., y(n-d_d). The parameters n_d and d_d represent the number of delay nodes associated with their respective input node. The second input could instead have been yn(n-1) and it's delayed values. The network has one hidden layer containing several hidden nodes that use a general output function, $f_j(\cdot)$. The output node uses a linear output function with a slope of one for scaling the output.

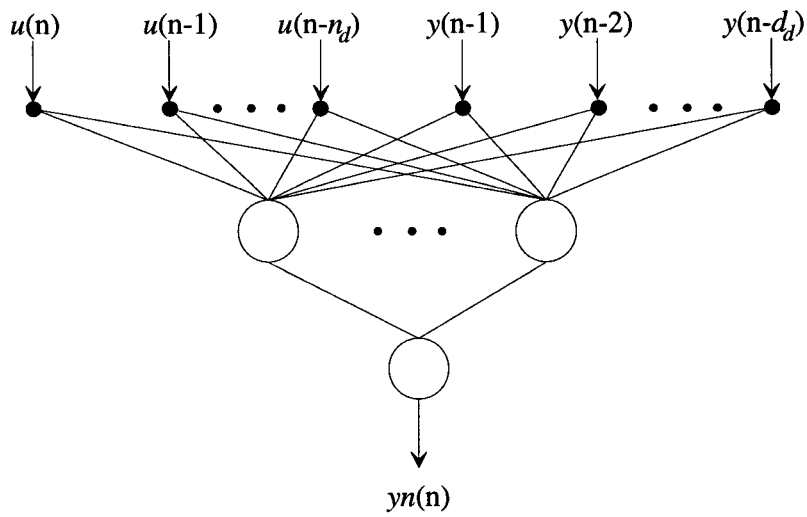

Figure 3. Multi-layer feedforward neural network with a time delayed structure

The equations for this network architecture is:

$$yn(n) = \sum_{j=1}^{hid} w_j \, f_j\big(net_j(n)\big) + b \tag{4}$$

and

$$net_j(n) = \sum_{i=0}^{n_d} w_{j,i+1} \, u(n-i) + \sum_{i=1}^{d_d} w_{j,n_d+i+1} \, y(n-i) + b_j \tag{5}$$

where

yn(n) is the output of the neural network,
$f_j(\cdot)$ is the output function for the jth node of the hidden layer,
net_j(n) is the activation level of the jth node's output function,
hid is the number of hidden nodes in the hidden layer,
n_d is the number of input nodes associated with $u(\cdot)$ not counting u(n),
d_d is the number of input nodes associated with $y(\cdot)$,
w_j the weight connecting the jth hidden node to the output node,

$w_{j,i}$ the weight connecting the ith input node to the jth hidden node,
y(n-i) is the delayed output of the plant used as an input to the network,
u(n-i) is the input to the network and its delays,
b_j the bias on the jth hidden node,
b the bias on the output node.

Prediction Using A Neural Network

The NGPC algorithm uses the output of the plant's model to predict the plant's dynamics to an arbitrary input from the current time, n, to some future time, $n+k$. This is accomplished by time shifting equations (4) and (5) by k, resulting in

$$yn(n+k) = \sum_{j=1}^{hid} \left\{ w_j \, f_j\left(net_j(n+k)\right) \right\} + b, \tag{6}$$

and

$$\begin{aligned} net_j(n+k) = & \sum_{i=0}^{n_d} w_{j,i+1} \begin{cases} u(n+k-i) & ,k-N_u < i \\ u(n+N_u) & ,k-N_u \geq i \end{cases} \\ & + \sum_{i=1}^{\min(k,d_d)} \left(w_{j,n_d+i+1} \, yn(n+k-i) \right) + \sum_{i=k+1}^{d_d} \left(w_{j,n_d+i+1} \, y(n+k-i) \right). \\ & + b_j \end{aligned} \tag{7}$$

The complexity of (7) arises when we take into account the cost function, J, and the recursive nature of the prediction. The first summation of (7) breaks the input into two parts represented by the conditional. The condition where k-N_u<i handles the previous future values of u up to $u(n+N_u-1)$. The condition where k-N_u>i sets the inputs from $u(n+N_u)$ to $u(n+k)$ equal to $u(n+N_u)$. The second condition will only occur if $N_2>N_u$. The next summations of (7) handles the recursive part of prediction. This feeds back the network output, yn, for k or d_d times, which ever is smaller. The last summation of (7) handles the previous values of y. The following example pictorially represents prediction using a neural network.

<u>Example</u>
Consider a network with two hidden nodes (h_d=2), one output node, and input consisting of u(n) and two previous inputs (n_d=2), and of three previous outputs (d_d=3). 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 conveyed in Figure 4.

The example below depicts a prediction of the plant for k=2. To produce the output yn(n+2), 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)] and [y(n) y(n-1) y(n-2)] 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 4. The network feedback is displayed as one network feeding another. This example shows the recursive nature of prediction.

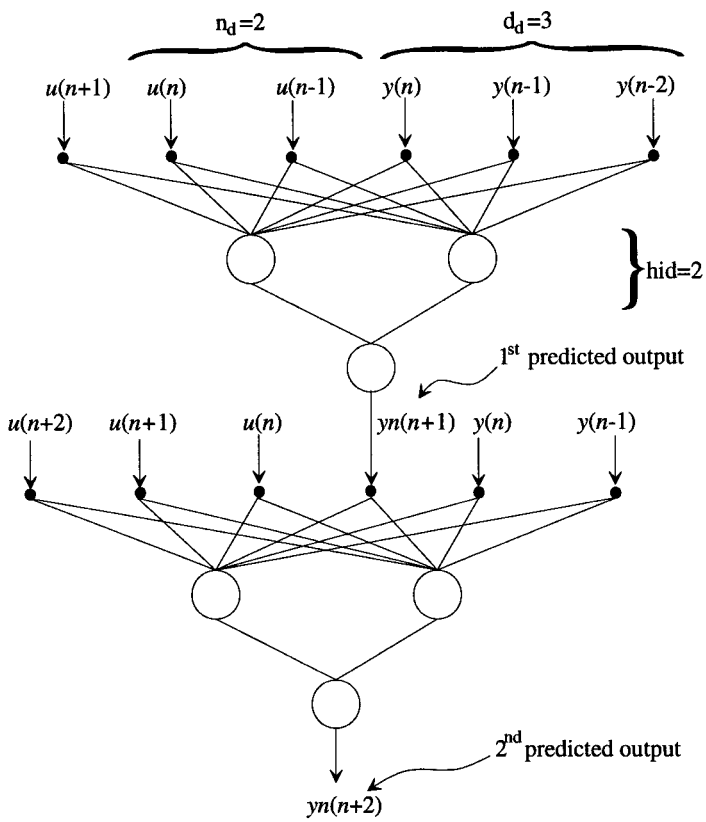

Figure 4. Network Prediction for k=2

Neural Network Derivative Equations

To minimize a cost function with a gradient based iterative solution in real-time, a computationally efficient derivation of the neural network's gradient is desired. The following is a derivation of the network's gradient equations.

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 (6) with respect to *u*(*n*+*h*) resulting in

$$\frac{\partial yn(n+k)}{\partial u(n+h)} = \sum_{j=1}^{hid} w_j \frac{\partial f_j\left(net_j(n+k)\right)}{\partial u(n+h)}. \tag{8}$$

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

$$\frac{\partial f_j\left(net_j(n+k)\right)}{\partial u(n+h)} = \frac{\partial f_j\left(net_j(n+k)\right)}{\partial net_j(n+k)} \frac{\partial net_j(n+k)}{\partial u(n+h)}, \tag{9}$$

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

$$\begin{aligned}\frac{\partial net_j(n+k)}{\partial u(n+h)} &= \sum_{i=0}^{n_d} w_{j,i+1} \begin{cases} \delta(k-i,h) & ,k-N_u < i \\ \delta(N_u,h) & ,k-N_u \geq i \end{cases} \\ &+ \sum_{i=1}^{\min(k,d_d)} w_{j,i+n_d+1} \frac{\partial yn(n+k-i)}{\partial u(n+h)} \delta_1(k-i-1)\end{aligned}. \tag{10}$$

Note that in the last summation of (10) the step function, δ_1, was introduced. This was added to point out that this summation evaluates to zero for k-i<1, thus the partial does not need to be calculated for this condition. The elements of the Hessian are obtained by differentiating Equations (8),(9), and (10) by $u(n+m)$, resulting in

$$\frac{\partial^2 yn(n+k)}{\partial u(n+h)\partial u(n+m)} = \sum_{j=1}^{hid} w_j \frac{\partial^2 f_j\left(net_j(n+k)\right)}{\partial u(n+h)\partial u(n+m)}, \tag{11}$$

$$\begin{aligned}\frac{\partial^2 f_j\left(net_j(n+k)\right)}{\partial u(n+h)\partial u(n+m)} &= \frac{\partial f_j\left(net_j(n+k)\right)}{\partial net_j(n+k)} \frac{\partial^2 net_j(n+k)}{\partial u(n+h)\partial u(n+m)} \\ &+ \frac{\partial^2 f_j\left(net_j(n+k)\right)}{\partial net_j(n+k)^2} \frac{\partial net_j(n+k)}{\partial u(n+h)} \frac{\partial net_j(n+k)}{\partial u(n+m)}\end{aligned}, \tag{12}$$

and

$$\frac{\partial^2 net_j(n+k)}{\partial u(n+h)\partial u(n+m)} = \sum_{i=1}^{\min(k,d_d)} w_{j,i+n_d+1} \frac{\partial^2 yn(n+k-i)}{\partial u(n+h)\partial u(n+m)} \delta_1(k-i-1).$$

Note that Equation (12) is the result of also applying the chain rule twice.

Simulation Results

Many physical plants exhibit nonlinear behavior. These relationships may be approximated by linear models, but often a nonlinear model would be more desirable. Nonlinear systems exhibit many phenomena that are not seen in linear systems such as an asymmetry between a step increase and decrease and the law of superposition does not hold. This section discusses the results from training a neural network to model a nonlinear plant and then using this model for NGPC. Later a comparison is made between control using the nonlinear model and a linear model.

Duffing's equation is a well studied nonlinear system. It can be thought of as representing the relationships between a mass, damper, and a stiffening spring. In

Equation (13), a non-minimum phase zero is added to demonstrate NGPC's capability to control a non-minimum phase plant,

$$\ddot{y}(t)+\dot{y}(t)+y(t)+y^3(t)=2u(t)-\dot{u}(t) \tag{13}$$

This plant was simulated using Runga-Kutta 4th order integration algorithm with a step size of 0.01 seconds and a sampling rate of 0.1 seconds/sample. Below, Figure 5 and Figure 6 show a series of pulses with increasing amplitude and the response of the plant due to this pulse train. 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 amplitude and that a frequency shift is also produced.

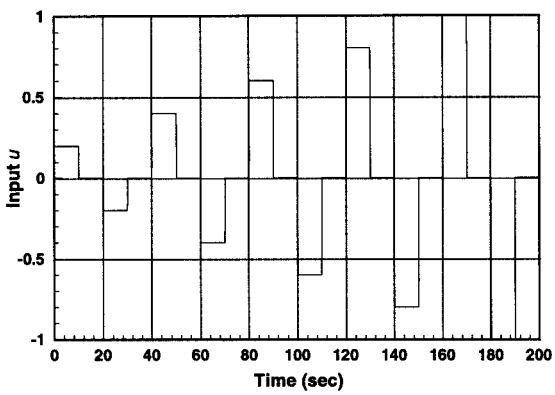

Figure 5. Pulse train used as the input to the plant

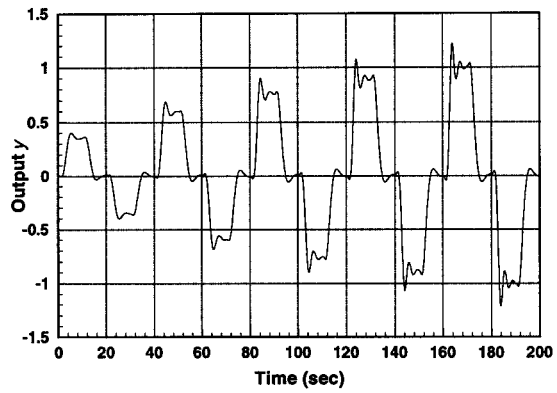

Figure 6. Response of plant due to pulse train input

The neural network was trained to model the plant using the same pulse train. The network architecture contains a single hidden layer with four hidden nodes and a single output node. The input layer is composed of two inputs, one that is externally fed with four time delayed nodes and the other is fed back from the output of the plant with four time delayed nodes. The hidden layer's nodes use the hyperbolic tangent as an activation function and the output is scaled linearly. Normalized Root Mean Squared error (NRMS) and Max error were used to measure training of the network. Equation (14) is the NRMS error measure, where T is the desired target output, O is the output of the network, p is the training pattern, and n is the output node. In this example, n is 1 and p is 2000 (200 time steps sampled at 0.1 seconds).

$$\mathbf{NRMS} = \sqrt{\frac{\sum_n \sum_p (T_{np} - O_{np})^2}{\sum_n \sum_p (T_{np})^2}} \tag{14}$$

Max error is the measure of the maximum error between T and O for each cycle. A cycle is defined as all of the input/output relations that form the entire pulse train. The NRMS error and the Max error for 10,000 cycles of network training are shown in Figure 7 and Figure 8.

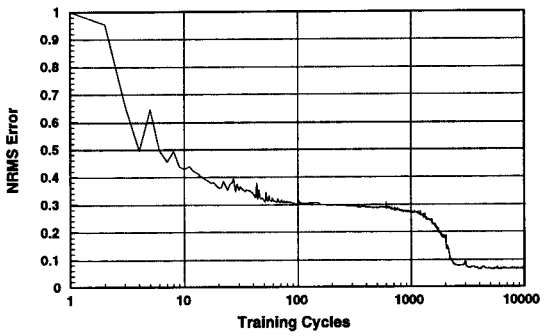

Figure 7. NRMS Error

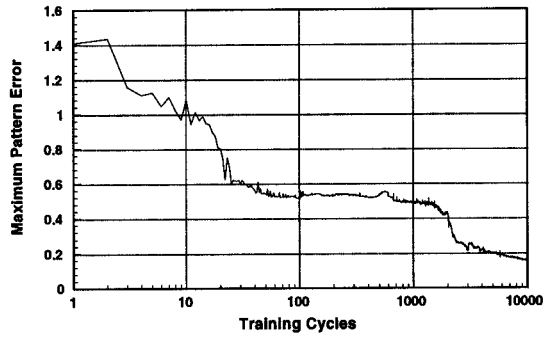

Figure 8. Max Error

The response of the plant is compared to the response of the trained network in Figure 9. The corresponding error between the plant and the network is shown in Figure 10.

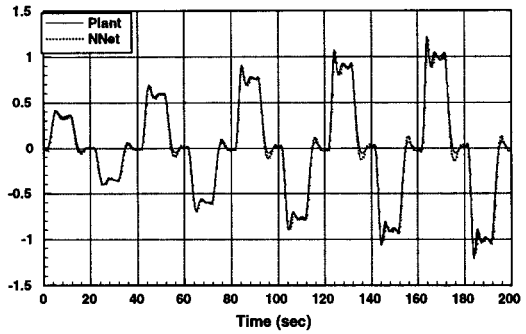

Figure 9. Response of the plant and network to pulse train input

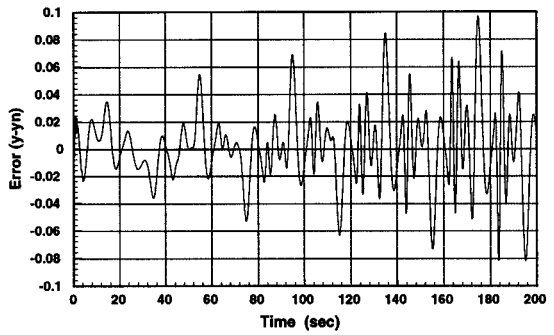

Figure 10. Error between plant and trained network

The nonlinear model of the plant was placed into the NGPC control-loop where the reference model is the pulse train filtered with a third order system with repeated poles at 0.3 rad/sec. The system was tuned by varying N_1, N_2, N_u, and λ to produce a desirable response. No input constraints were used in this example. The final tuning resulted in N_1=1, N_2=17, N_u=1, and λ=0. The controlled response of the plant tracking the filtered pulse train is shown in Figure 11. The corresponding error between the plant and the reference model is shown in Figure 12.

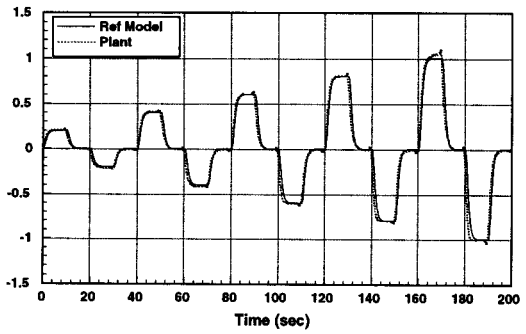

Figure 11. Plant tracking using the neural network as the plant's model

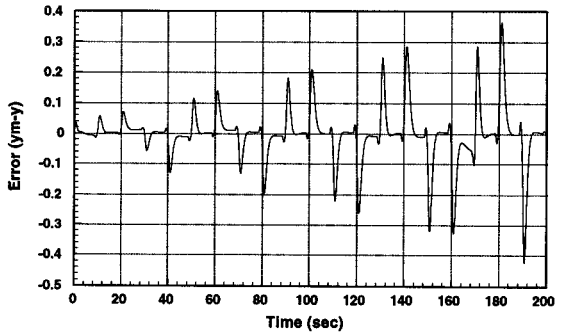

Figure 12. Error between reference model and plant

To compare these results to results where the model is linear, equation (13) was linearized about zero. Discretizing with a sample time of 0.1 seconds using the step invariant transform produces

$$y(n)-1.895y(n-1)+0.9048y(n-2)=-0.0853u(n-1)+0.1044u(n-2). \tag{15}$$

Equation (15) replaced the neural network as the plant estimator. Figure 13 shows the controlled response of the plant with the linear plant as the model. The corresponding tracking error is shown in Figure 14. Comparing results shown in Figure 11 and Figure 13 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.

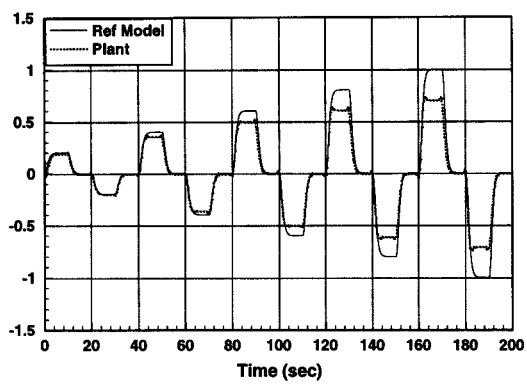

Figure 13. Plant tracking the reference model that used linear model of the plant

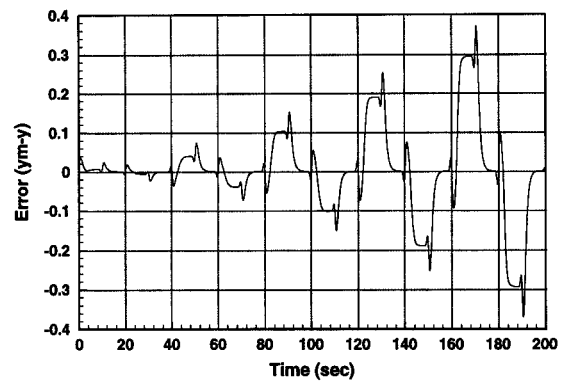

Figure 14. Error between reference model and plant

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 discuss the various parameters that determine the process speed for NGPC.

The algorithm, in conjunction with the complexity of the plant, determines the rate of convergence. In this paper the Newton-Raphson optimization has been implemented and for various plants it has been found to converge to a good result within two iterations[2]. The plants all 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.

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

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 will save a considerable amount of CPU time.

Another component of process speed is the computer's performance. To ensure optimal speed the computer, compiler, and floating point data types were all 32 bit. A Pentium Pro 150 MHz PC, 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.

Timing Specifications

The previous section outlined many factors that determine the process speed. To demonstrate the servo rates that one might expect to get with this algorithm and the optimizations, timing data was collected using a simulated plant for different values of N_2 and N_u. The neural network had two inputs with four and five delays on the first and second input respectively, six hidden nodes and one output node. The data shown in Table 1 represents servo rates in Hertz where the number of iterations taken to produce a solution was set to two. This would be a typical setting for real-time applications. The values for N_2 ranged from 1 to 20 and the values for N_u varied from 1 to 10. Note that the Table 1 is lower triangular because N_u cannot be larger than N_2.

N_2 \ N_u	1	2	3	4	5	6	7	8	9	10
1	8596									
2	5579	4378								
3	4104	3138	2589							
4	3238	2405	1900	1597						
5	2655	1936	1487	1211	1035					
6	2232	1609	1210	964	801	695				
7	1920	1366	1010	789	643	544	482			
8	1692	1188	870	670	537	448	388	350		
9	1511	1052	762	583	463	380	325	288	262	
10	1366	945	683	516	407	331	279	243	218	201
11	1246	857	615	462	362	292	244	211	187	170
12	1145	783	558	418	325	262	218	187	164	148
13	1058	720	511	382	297	237	197	168	147	131
14	984	667	472	351	272	218	180	153	133	117
15	920	622	440	326	252	201	165	140	121	107
16	865	582	411	304	234	186	152	129	111	98
17	815	548	386	284	219	173	142	120	103	90
18	771	517	364	268	206	163	133	112	95	83
19	731	490	344	253	194	153	125	104	89	77
20	695	465	326	239	184	145	118	98	83	72

Table 1. Timing Data for NGPC where N_2 and N_u are varied

The cost of the NGPC algorithm can be broken down into five separate cost. The Jacobian, Hessian, plant prediction, LU decomposition, and other miscellaneous over head are calculated based on a percentage of computational cost. The case where both N_2 and N_u are five is presented in Table 2.

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 2. Percentage of Time for Key Routines where N_2=5 and N_u=5

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 [15]. 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 [15] takes 10 to 20 iterations, the Newton-Raphson algorithm runs 6.1 to 12.2 times faster.

Conclusions

This paper has developed a computationally efficient Neural Generalized Predictive Controller utilizing Newton-Raphson optimization algorithm to minimize the GPC cost function with input constraints. The simulation results showed NGPC improved control performance over GPC with a linear model. The real-time capability of this algorithm was demonstrated by presenting timing data from a case study that showed that a typical servo rate of 1000 Hz is attainable with a Pentium Pro 150 MHz PC.

References

[1] D. W Clarke and C. Mohtadi, "Properties of Generalized Predictive Control", *Automatica* Volume 25, Issue 6, pp. 859-875, 1989.

[2] D. E. Rumelhart, J. L. McClelland and the PDP Research Group, "Parallel Distributed Processing", Volume 1, Chapter 8, *The MIT Press* 1986.

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

[4] D. W. Clarke, C. Mohtadi and P. C. Tuffs, "Generalized Predictive Control - Part 1: The Basic Algorithm," *Automatica*, Volume 23, pp. 137-148, 1987.

[5] D. W. Clarke, C. Mohtadi and P. C. Tuffs, "Generalized Predictive Control - Part 2: The Basic Algorithm," *Automatica*, Volume 23, 1987, pp 149-163.

[6] Y. C. Jung, and R. A. Hess, "Precise Fight-Path Control Using a Predictive Algorithm", *Journal of Guidance, Control, and Dynamics*, Vol 14, 1991 pp 936-942.

[7] 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.

[8] H. Domirciogiu, and D. W. Clarke, "CGPC with Guaranted Stability Properties," *IEE Proceedings-D*, Vol. 139, No. 4, July 1992.

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

[10] Jeong Jun Song and Sunwon Park, "Neural Model-Predictive Control for Nonlinear Chemical Processes," *Journal of Chemical Engineering of Japan*, 1993, V26, N4, p347-354.

[11] 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.

[12]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.

[13] 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.

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

[15] 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.

[16] Jeong Jun Song and Sunwon Park, "Neural Model-Predictive Control for Nonlinear Chemical Processes," *Journal of Chemical Engineering of Japan*, 1993, V26, N4, p347-354.

[17] D. C. Psichogios and L. H. Ungar, "Nonlinear Internal Model Control and Model Predictive Control using Neural Networks", *Proceedings. 5th IEEE International Symposium on Intelligent Control 1990*, 5-7 Sept. 1990, Philadelphia, PA, USA, pp1082-1087.

[18] 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.