AIAA-98-1779

NEURAL NETWORKS FOR RAPID DESIGN AND ANALYSIS

Dean W. Sparks, Jr.* and Peiman G. Maghami†

NASA Langley Research Center, Hampton, VA 23681-0001

Abstract

Artificial neural networks have been employed for rapid and efficient dynamics and control analysis of flexible systems. Specifically, feedforward neural networks are designed to approximate nonlinear dynamic components over prescribed input ranges, and are used in simulations as a means to speed up the overall time response analysis process. To capture the recursive nature of dynamic components with artificial neural networks, recurrent networks, which use state feedback with the appropriate number of time delays, as inputs to the networks, are employed. Once properly trained, neural networks can give very good approximations to nonlinear dynamic components, and by their judicious use in simulations, allow the analyst the potential to speed up the analysis process considerably. To illustrate this potential speed up, an existing simulation model of a spacecraft reaction wheel system is executed, first conventionally, and then with an artificial neural network in place.

Introduction

The overall design process for aerospace systems typically consists of the following steps: design, analysis and evaluation. If the evaluation is not satisfactory, the process is repeated until a satisfactory design is obtained. Dynamics and control analyses, which define the critical performance of many aerospace systems, are particularly important. Generally, all aerospace systems experience excitations resulting from internal and external disturbances, for example, aerodynamic turbulence encountered by aircraft or instrument scanning in space systems. Excessive vibrations due to turbulent aerodynamics could diminish the ride quality or safety of an aircraft. In space systems, excessive vibrations could be detrimental to its science instruments which usually require consistently steady pointing in a specified direction for a prescribed time duration. Typically, in the course of the design of an aerospace system, as the definitions and the designs of the system and its components mature, several detailed dynamics and controls analyses are performed in order to insure that all mission requirements are being met. These analyses, although necessary, have historically been very time consuming and costly due to the large number of disturbance scenarios involved, and the extent of time domain simulations that need to be carried out. For example, a typical pointing performance analysis for a space system might require several months or more, which can amount to a considerable drain on the time and resources of a space mission.

It is anticipated that artificial neural networks (ANNs) can be used to significantly speed up the design and analysis process of aerospace systems. This paper will focus on the application of ANNs in approximating nonlinear dynamic components in simulations, in order to reduce overall time domain analysis time and compute effort. Initial work has shown that ANNs, once properly trained, can be used in place of nonlinear dynamical systems in simulations. These ANNs can give very good approximations of the systems' outputs, and they can drastically reduce computational burden in running the overall simulation. A numerical example of a dynamical system simulation with an ANN is presented, and comparisons between conventional (i.e., without the ANN) simulation times, in terms of computer processing unit (CPU) seconds, versus simulation times with the ANN in place is made.

The paper is organized as follows. After this introduction section, a brief description on conventional dynamics analysis is given. Next, discussions on

---

* Aerospace Technologist, Guidance and Control Branch.

† Senior Research Engineer, Guidance and Control Branch, Senior Member, AIAA.

neural networks, their use in approximating functional relationships, together with a typical design outline, is presented. Then, numerical results of an example application of an ANN in a simulation is reported. Finally, a conclusions section closes the paper.

Conventional Dynamics Analysis

Conventional dynamics analysis can be divided into two categories: time domain analysis and frequency domain analysis. Both are used to determine specific characteristics of a system performance, but as implied by their respective names, the characteristics are either defined in terms of time or as a function of frequency. In this paper, the emphasis will be on time domain analysis. Time domain analysis tries to compute the transient and steady state time responses of a system given specific inputs. Examples of typical system response characteristics, which are studied in time domain analysis, include transient system response maximum overshoot, rise and settling times. Another is the system's steady state performance, which is usually defined by some metric on the steady state error between the system response and a reference signal. If the system is simple enough, i.e., linear, of very low order and has relatively few inputs and outputs, like a single-input, single-output (SISO) system, its responses can be obtained by direct solution of the system equations which describe the model. However, most realistic system models are of high order and/or nonlinear, which precludes a direct solution. The usual procedure in this case is to construct a simulation of the system to obtain the time responses, via integration (e.g., Runge-Kutta methods) of the system's equations of motion. There are available several simulation-based packages, such as MATRIXx/System Build and MATLAB/Simulink, which can perform whatever time domain analysis is required. However, even with these tools, computing time response solutions can be expensive both in terms of time and effort, depending upon a number of factors, such as the order of the system, the number of inputs and outputs, the level of nonlinearities, the type and level of disturbance inputs and/or reference signals, and the kind of integration selected.

Whatever type of analyses need to be done, it would be highly beneficial to the analyst to be able to rapidly assess the effects on system time response performance due to the almost inevitable design changes that a system will undergo during its lifetime. During the design phase of an aerospace system, almost all components go through some level of change, with each change having the potential to affect the performance of the overall system to some degree. In many instances, these changes are expected to affect the performance of the system to such a degree as to warrant a partial or full analysis of its performance. In the area of spacecraft dynamics and controls, these types of changes include: changes in the inertia or flexibility of the structural components which would affect the dynamic characteristics of the spacecraft; changes in the characteristics of the external and internal disturbances that may act on the spacecraft while it is in orbit; or changes in the control system design, hardware, and software. For example, for a reaction wheel system, changes could include: wheel size, nonlinear friction characteristics, or wheel speed internal controller design. Now, depending on the nature and extent of these changes, there may be a need to reevaluate the controlled dynamical responses of the system. The computational time and cost associated with each of these performance analyses (i.e., executing conventional time simulations) may be substantial. The cost can be exorbitant especially if the analysis has to be repeated several times during the design phase. One approach to this problem is to use artificial neural networks (ANNs) to help speed up the analysis.

Rapid Analysis with ANNs

The motivation behind the use of ANNs is to speed up the analysis process substantially. The main use of ANNs lies with their ability to approximate functional relationships, specifically nonlinear relationships. This can be either a static relationship, one that does not involve time explicitly, or a dynamic relationship, which explicitly does involve time. Dynamic approximations via ANNs can be achieved by using the appropriate time delays and feedback of the output back to the input, which is defined as recurrence. Such networks are referred to as recurrent networks[1,2]. In any case, to an ANN, there is no distinction between a static or dynamic map, there is just input/output data. For example, an ANN could be designed to approximate the dynamic behavior of a nonlinear component, e.g., the mapping between the nonlinear torque output of a spacecraft reaction wheel and its angular wheel speed and input torque command. Once such a network is trained, the torque output of the wheel, for given wheel speed and torque command inputs, can be easily obtained by simulating the ANN. One application of ANNs is to use them to speed up the simulation process and therefore, the overall analysis time. For example, ANNs can be designed to approximate the outputs of a continuous-time, nonlinear system, with outputs computed for a specified discrete step. This way, the

traditional continuous-time integration (e.g. Runge-Kutta) of the nonlinear dynamics can be replaced by discrete-time nonlinear algebraic updates, with reasonable accuracy. Although the initial training time for an ANN may be long, it can be performed during off hours, in a semi-automated manner, without much direct involvement by the designer. Also, once an ANN has been designed to represent a dynamic component, it can be stored in a component library and recalled for use in future analyses.

The successful design of an ANN depends on the proper training of the network. The training of a network involves the judicious selection of points in the input variable space, which along with the corresponding output points, constitute the training set. In the reaction wheel example, in order to properly train an ANN approximation, it is important that the input points, i.e., the wheel speed and commanded torque values, completely cover the range of possible values for both. In addition, it is important that enough points are selected such that they cover areas where fine resolution in the design space is required, i.e., areas where small variations in input data cause large variations in the corresponding output data. Of course, there will be the inevitable trade-off between selecting enough points for good training and keeping the number of training points down to practical levels for computation.

Before proceeding, it is important to restate here that the true advantage of using ANNs lies with modeling nonlinear relationships. Although one can certainly use ANNs to represent linear systems, there will be no gain, in terms of reductions in compute time and effort, in their use over conventional representations of the same linear systems. One can always take any pure linear, dynamical system and rewrite it as a series of output difference equations, which are functions of appropriate time-delayed output feedbacks and input signals. It turns out that the coefficients of these system output equations are equivalent to the "weighting coefficients" (which are defined in the following subsection) of pure linear ANNs, with the "bias" parameters (see following subsection) set to zeros. Thus, there would be no point in training ANNs to represent linear dynamical systems. Therefore, the work reported in this paper will only cover representing nonlinear systems with ANNs.

In the following subsections, a brief overview of ANNs and the training of a specific type of ANN that was used in this work, are presented.

Artificial Neural Networks (ANNs)

Artificial neural networks (ANNs) have grown into a large field since their inception, and a complete discussion on them is beyond the scope of this paper. Instead, this section will present a very brief description on ANNs. ANNs were developed as an attempt to mimic the process of the human brain. They consist of groups of elements (called neurons) which perform specific computations on incoming data, with interconnections which permit data flow from one group of neurons to the next, similar to the way groups of biological neurons receive and transmit information through dendrites and axons, respectively, in a brain. Like their biological counterparts, ANNs can be trained to perform a variety of tasks, such as modeling functional relationships. The parameters of the ANN, when presented with the appropriate input and output data related to a specific functional relationship, can be adjusted such that the ANN can give a good representation of that relationship. This feature is particularly useful when the relationship is nonlinear and/or not well defined, and thus difficult to model by conventional means. Also ANNs, by their very nature, are a perfect fit for efficient parallel computations on digital computers. Though there are several types of ANNs, in this paper, only the feedforward ANN will be discussed.

A typical feedforward ANN is depicted in Figure 1, with m inputs and n_p outputs, and each

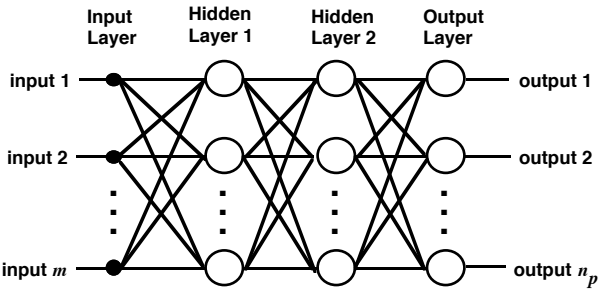

Figure 1. Typical feedforward ANN.

circle, or node, representing a single neuron. The name feedforward implies that the data flow is one way (forward) and there are no feedback paths between neurons. The output of each neuron from one column is an input to each neuron of the next column. Using the typical naming convention, each column of neurons is called a layer, the initial column where the inputs come into the ANN is called the input layer, and the last layer, i.e., where the outputs come out of the ANN, is denoted as the output layer. All other layers in between are called hidden layers. These ANNs can have as many layers as desired, and each hidden layer can have

as many neurons as desired. Each neuron can be modeled as shown in Figure 2, with n being the number of inputs to the neuron.

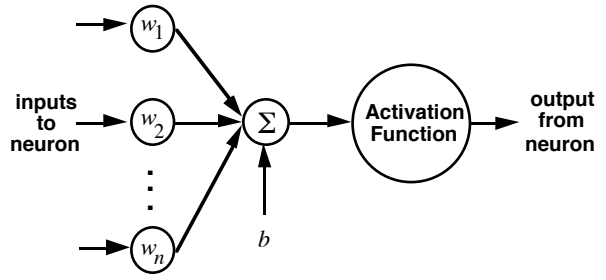

Figure 2. Representation of a neuron in the feedforward ANN.

Associated with each of the n inputs is some adjustable scalar weight, w_i , i = 1, 2, ..., n, which multiplies that input. In addition, an adjustable bias value, b, can be added to the summed scaled inputs. These combined inputs are then fed into an activation function, which produces the output of the neuron. The activation function can take on many forms to shape the output; three of the more common functions are linear, tan sigmoid, and log sigmoid, as shown in Figure 3. The

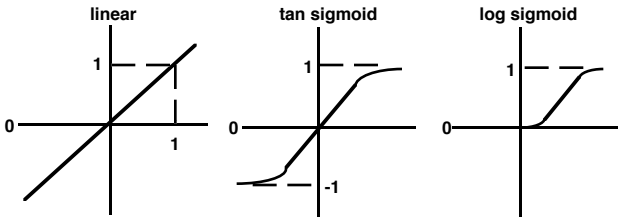

Figure 3. Three common activation functions.

linear activation function simply outputs the input; the tan sigmoid function is the hyperbolic tangent function, with output values between [-1,1] for inputs (-∞,+∞); while the log sigmoid is also a nonlinear function, which can be written as $y = 1/(1+e^{-x})$, with the outputs values, y, in the range [0,1], given inputs, x, in the range (-∞,+∞). During training, the set of weights and bias terms associated with the neurons are adjusted until the output of the ANN matches, to within some specified level of tolerance, the true outputs for the same inputs.

Training of a Feedforward Network

The objective is to design a feedforward network to map the functional relationship between a set of input points and a corresponding set of output points, or target points. To accomplish this task, a feedforward network, like the one shown in Figure 1, but with only one hidden layer, is considered. The input layer has n_c nodes, corresponding to the elements of the input vector, while the output layer has n_p nodes, which correspond to the elements in the output vector. The number of nodes in the hidden layer is arbitrary, however, it has to be large enough to guarantee convergence of the network to the functional relationship that it is to approximate. Once the number of nodes in the hidden layer has been chosen, the network design is reduced to adjusting, or training, the weighting coefficients and biases. The parameters of feedforward networks are usually trained using either a gradient method named the back propagation method[1,2], or a pseudo-Newtonian approach, such as the Levenberg-Marquardt[3] technique. Typically, in these methods, the weights and biases are trained to minimize some cost function of the error of the network. The network error is defined as the difference between the output of the true system and that of its ANN approximation, for a given set of inputs. The cost function is usually taken as the sum squared error of the network over all of the input points. If q sets of points (e.g., points taken for q time samples) are used for training the network, then the input U to the network would be an n_c x q matrix, with each column corresponding to a set of input points for a given time sample, and the output Y_p would be a n_c x q matrix, with each column of Y_p corresponding to that of U. Now the cost function, in terms of the sum squared error of the network, can be written as

$$E = \sum_{k=1}^{qn_p} e(k)^2 = \sum_{r=1}^{q}\sum_{j=1}^{n_p}(Y_d(j,r) - Y_p(j,r))^2 \qquad (1)$$

where Y_d is a n_p x q matrix of the target outputs. The typical procedure is to keep updating the weights and biases until the error E goes below some specified tolerance level. At this point, the feedforward network is considered trained.

It has been shown in the literature that a feedforward network with only one hidden layer can approximate a continuous function to any degree of accuracy[4-6]. It is obvious that this capability carries over to networks with more than one hidden layer. The use of feedforward ANNs has some advantages over the conventional approximation techniques, such as polynomials and splines. For example, polynomials are hard to implement in hardware due to signal saturation, and if they are of higher order, there may be stability problems in determining the coefficients. ANNs, on the other hand, are very amenable to hardware implementation. As a matter of fact, to date, several VLSI chips based on multilayer neural network architecture are available[7,8].

Reaction Wheel Model Example

In order to illustrate the feasibility of using ANNs to approximate dynamic components, a model of a reaction wheel assembly, consisting of three reaction wheels, one each for the roll, pitch, and yaw axes of a spacecraft, was selected as a test application. Figure 4 shows the block

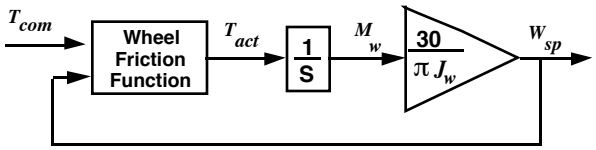

Figure 4. Example Reaction Wheel Model.

representation of a reaction wheel model in this assembly; this model was used for all three wheels. This model is fairly simple in nature, and consists of the following: the input, T_{com}, is the torque command (in units of N-m) to the reaction wheel, which is updated every 1.024 seconds; T_{act} , in N-m, is the actual torque output of the wheel, which includes nonlinear viscous friction torque; the wheel momentum, M_w; and the angular wheel speed, W_{sp}, which is converted into units of revolutions per minute (RPM). The parameter J_w is the wheel inertia. The actual torque output of the wheel, T_{act}, is the combination of the torque command and viscous friction torque, T_{fric} , (which takes the opposite sign of that of the wheel speed W_{sp}):

$$T_{act} = T_{com} - T_{fric} * sign(W_{sp}). \qquad (2)$$

Two different nonlinear functions were used to model the wheel friction torque: a quadratic function in terms of wheel speed; and an exponential function in terms of wheel speed. In each case, an ANN was designed to approximate the dynamics of the wheel. The results are presented in the following subsections.

Quadratic Friction Function

In the first case, the wheel viscous friction torque was modeled with a quadratic function in terms of W_{sp}, which is given below:

$$T_{fric} = 3.367176\text{x}10^{-5} * aw + 2.41045\text{x}10^{-6} * aw^2, \quad aw = |W_{sp}|. \qquad (3)$$

The above wheel model is continuous and nonlinear, and in the past, has been simulated using a Runge-Kutta (2,3) variable step size integration for accurate, but time consuming, integration. The error tolerance for the integration was set at 10^{-6}, the minimum step size set at 10^{-8} seconds, and the maximum step size at one time sample of 1.024 seconds. Note that the tight error tolerance was required for solution accuracy. One way to speed up the simulation was to convert the continuous-time model into a discrete-time model, and then use discrete updates at every 1.024 seconds to propagate the system state equations. However, as will be discussed later in this section, the direct discrete simulation of this model results in unacceptable inaccuracies because of the nonlinear torque friction component.

To try to keep the speed advantage of discrete update simulations, and still maintain reasonable accuracy in the wheel model outputs, an ANN was trained to map the functional relationship from the torque input command at the *k*th discrete time step, $T_{com}(k)$, and wheel speed at the *k*th time step, $W_{sp}(k)$, to the wheel speed for the next time step, $W_{sp}(k+1)$. In other words, the wheel speed from one time step to the next was approximated. Figure 5 depicts the discrete-time model of the reaction wheel, with a single-hidden layer ANN (hidden layer with a tan sigmoid activation function, the output layer with a pure linear function) computing $W_{sp}(k+1)$. A unit delay is in place to obtain the current (*k*th step) wheel speed.

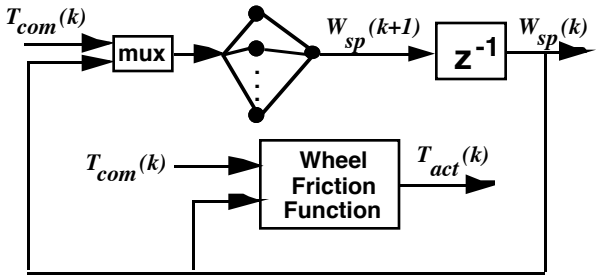

Figure 5. Discrete reaction wheel model with an ANN.

The friction torque computation was done the same way as in Figure 4. It was felt that there would be no real advantage gained in substituting a separate ANN to replace the simple quadratic friction function (Eq. 2), which was a static map. Since the same model was used for each of the three wheels in the assembly, the same ANN could be used for each wheel.

Before the ANN for the wheel speed could be trained, the appropriate input/output training data had to be generated. To accomplish this, proper data points for both the torque command T_{com} and wheel speed W_{sp} , the two inputs to the ANN, had to be selected first. With this specific wheel model, the expected operating range for T_{com} was assumed to be +/- 0.1 N-m, and +/-

300 RPM for W_{sp}. To get adequate coverage of data points over these ranges, the T_{com} points were taken in equally-spaced increments of 0.005 N-m, while the W_{sp} points were taken in increments of 3.0 RPM. With these ranges and increments, the total number of training input pairs (T_{com}, W_{sp}) was 8,241. The corresponding training output, or target, points were then computed by taking each training input pair, and running a MATLAB (v5.0)/Simulink (v2.0) simulation of the continuous reaction wheel model (Figure 4) over a specified time interval [0, T_{step}]; the wheel speed value at time T_{step} was recorded as the desired target point for that specific training pair. As mentioned earlier, for this reaction wheel model, T_{step} was set to 1.024 seconds. Each Simulink simulation used the second-order, three-function-evaluation-per-step Bogaki-Shampine variable-step integration routine, the minimum and maximum step sizes allowed were set at 10^{-8} and 1.024 seconds, respectively; the relative and absolute error tolerance parameters were set to 10^{-6}. Each T_{com} input value was held constant over the integration range [0, 1.024], while the corresponding W_{sp} input value was entered as the initial wheel speed value (i.e., at time 0) in the pure integrator block.

After the training data was generated, the ANN could now be trained. Prior to the actual ANN training, both the input and output training data were normalized with respect to their absolute maximum values; by keeping the training data in the [-1, 1] range, more efficient use of the ANN training routines was obtained. The training led to a feedforward ANN with one 10-neuron, hidden layer (using a tan sigmoid activation function) and a pure linear output layer. The training was performed using the standard 'trainlm' function from the MATLAB Neural Network Toolbox, which is based on the Levenberg-Marquardt training algorithm [Ref. 3]. Running on a Sun Ultra-2 Workstation, the training of this ANN completed in less than 2.0 (elapsed time) hours. The training reduced the sum squared error (see Eq. (1)) of the ANN down to a level of 2.98 x 10^{-5}, which was deemed acceptable. Once the training was completed, the final ANN weights and bias numbers were scaled back to their true values. In checking the accuracy of the approximation achieved by this ANN, given the training input points, the mean percent error between the true target points and the ANN outputs points was 0.063% , and only 1.07% of the points had errors greater than 1%.

Once the ANN-based model of the reaction wheel assembly was developed, its performance, in terms of accuracy and execution CPU time, was evaluated in several discrete-time simulations under a specific set of torque command input, T_{com}, profiles. These 6000-second torque command profiles for the roll, pitch and yaw axis wheels, respectively, shown in Figure 6, were a series of 1.024-second-wide pulses. In addition, a small random signal was added to the pitch axis wheel torque command. These could be typical torque command profiles required to counter the motions of

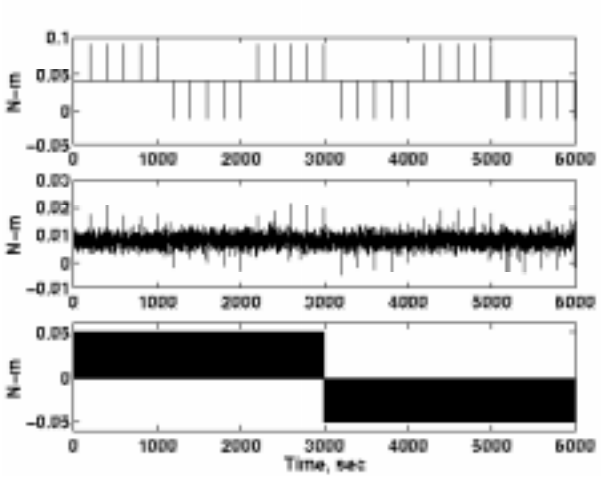

Figure 6. Roll, pitch and yaw torque input command profiles.

scanning instruments on a spacecraft, for example. The original continuous-time reaction wheel assembly model, as shown in Figure 4, was also simulated using the Simulink second-order, three-function-evaluation-per-step Bogaki-Shampine variable-step integration routine, with the minimum and maximum step sizes allowed were set at 10^{-8} and 1.024 seconds, respectively. The relative and absolute error tolerance parameters were set to 10^{-6}. The results of this simulation were considered the 'true' results, against which the other simulations were tested. The average execution time for this 'true model' simulation was 15.01 CPU seconds on the Sun Ultra-2.

Using the ANN-based model of the reaction wheel assembly, two different discrete-time simulations, both running at a discrete update period of 1.024 seconds, were performed to see if meaningful reductions in simulation execution times can be achieved without sacrificing accuracy down to unacceptable levels. Table 1 contains the execution times (in CPU seconds), the rms and maximum absolute errors (as compared to the 'true model' results from above) for three discrete-time simulations . First, a MATLAB function file version of the ANN-based discrete-time model was written; this function was executed in MATLAB v5.0. The average execution time for was 6.49 CPU seconds. In comparing the wheel speed outputs from this discrete function file simulation with those from the 'true model', very good agreement was observed; the

Table 1. Discrete-time simulation results for quadratic friction case.

Simulation	CPU sec	rms error (RPM)	Max. error (RPM)
ANN MATLAB function	6.49		
roll axis wheel		0.0011	0.0092
pitch axis wheel		0.0049	0.0790
yaw axis wheel		0.0263	0.0417
ANN MEX file	0.35		
roll axis wheel		0.0011	0.0092
pitch axis wheel		0.0049	0.0790
yaw axis wheel		0.0263	0.0417
discrete MATLAB function	1.71		
roll axis wheel		1.8437	41.5683
pitch axis wheel		1.3968	80.6037
yaw axis wheel		4.2915	63.6718

root-mean-square (rms) of the errors between 'true' and ANN-based discrete-time simulation outputs, over the length of the simulation, were 0.0011 RPM for the roll axis wheel, 0.0049 RPM for the pitch axis wheel, and 0.0263 RPM for the yaw axis wheel. These results indicated that, although acceptable simulation accuracies were achieved with the ANN-based model, the execution time speed up was only a factor of 2.3.

This was somewhat expected, because the friction nonlinearity was fairly benign, i.e., the Runge-Kutta integration did not have to take many steps to converge to the solution. More reduction in execution time can be achieved if another compute language is used, one with faster loop execution capability. To do this, the ANN-based wheel model simulation was written in FORTRAN-77, for execution as a MEX file called by MATLAB. MEX files are dynamically linked subroutines which MATLAB can load and execute like regular MATLAB functions.

The third simulation was just a pure discrete-time simulation (zero-order-hold integration, with no ANN) of the wheel assembly, sampled at 1.024 seconds. This simulation was also written as a MATLAB function, and executed in MATLAB v5.0.

The results in Table 1 show that although the pure discrete wheel model simulation executes at a faster rate, its accuracy leaves much to be desired. Figure 7. shows the roll axis wheel speed output time histories for this case, from the 'true model' Simulink simulation (top), from the ANN-based model MEX file simulation (middle), and the pure discrete-time model simulation (bottom). In these simulations, the initial angular speed of all three wheels was 250 RPM.

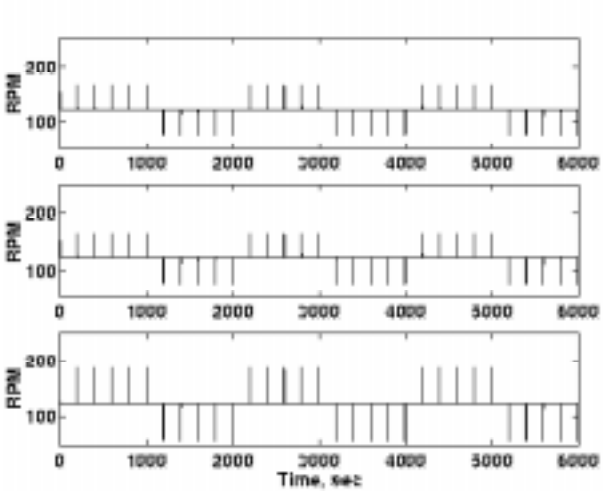

Figure 7. Roll axis wheel speed simulation results.

Clearly, the ANN-based MEX file simulation results matched the 'true model' results much better than did the pure discrete-time simulation results. The combination of the nonlinearity and the rapid dynamics caused by the pulse command profile made it difficult for the pure discrete model to accurately match the 'true model' simulation, at the update period of 1.024 seconds. On the other hand, the ANN-based wheel model simulations, while executing slower than the pure discrete model simulation, gave much more accurate results. The ANN-based wheel model MEX file simulation gave wheel output results which were very comparable to the 'true model' results, while executing about 40 times faster, which was a fairly significant speed up.

Exponential Friction Function

In the second case, the wheel viscous friction torque was modeled with an exponential function in terms of W_{sp}, which is given below:

$$T_{fric} = 0.01 * aw * e^{-0.01aw}, \quad aw = \left|W_{sp}\right|. \tag{4}$$

As in the quadratic friction function case, an ANN-based model of the reaction wheel assembly was designed. Another 10-node, feedforward ANN, was trained in the exact manner as reported in the previous case. Running on the Sun Ultra-2 Workstation, the training of this ANN completed in less than 2.0 (elapsed time) hours. The training reduced the sum squared error of this ANN down to 3.009×10^{-4}, which was deemed acceptable. In checking the accuracy of this ANN, given the training input points, the percent error between the true target points and the ANN outputs points were 0.419% on average, and only 1.32% of the

points had errors greater than 1%. It should be noted that the percentage numbers above did not include those target wheel speed points that were very close to zero magnitude, since they made the percent error calculations biasly inaccurate. In checking the absolute errors, between 'true' targets and ANN outputs for all 8,241 points, the mean error was 0.017 RPM, and the maximum absolute error was 0.479 RPM.

Again, the original continuous-time model of the reaction wheel assembly was simulated using the Simulink second-order, three-function-evaluation-per-step Bogaki-Shampine variable-step integration routine, with the minimum and maximum step sizes allowed set at 10^{-8} and 1.024 seconds, respectively. The relative and absolute error tolerance parameters were set to 10^{-6}. The results of this simulation were considered the 'true' results, against which the other simulations were tested. The average execution time for this 'true model' simulation was 65.54 CPU seconds; the higher magnitude and nonlinear nature of the exponential friction model caused the Simulink integration routine to take smaller time steps than with the quadratic friction model, thus the longer execution time.

Using the ANN-based model of the reaction wheel assembly, two different discrete-time simulations, both running at a discrete update period of 1.024 seconds, were performed for comparison with the 'true model' Simulink simulation. Table 2 contains the execution times (in CPU seconds) for three discrete-time simulations, and the rms and maximum absolute errors (as compared to the 'true' results from above) for three discrete-time simulations. First, the same MATLAB v5.0 function file version of the discrete-time ANN-based model, used in the previous case, was executed using the same torque input profiles. The average execution time was 6.5 CPU seconds. In comparing the wheel speed outputs from this discrete function file simulation with those from the 'true model' Simulink simulation, very good agreement was observed; the root-mean-square (rms) of the errors between 'true' and ANN-based discrete-time simulation outputs, over the length of the simulation , were 0.0007 RPM for the roll axis wheel, 0.0014 RPM for the pitch axis wheel, and 0.0018 RPM for the yaw axis wheel. The advantage of an ANN-based, discrete-time wheel model was really brought out in this case with the higher nonlinear friction model. The ANN-based model MATLAB function simulation executed 10 times faster than its corresponding 'true model' Simulink simulation, with excellent results.

Again, to see if even more execution speed-up could be achieved, the ANN-based model simulation was written as a FORTRAN-77/MEX file.

Table 2. Discrete-time simulation results for exponential friction case.

Simulation	CPU sec	rms error (RPM)	Max. error (RPM)
ANN MATLAB function	6.50		
roll axis wheel		0.0007	0.0139
pitch axis wheel		0.0014	0.0102
yaw axis wheel		0.0018	0.0068
ANN MEX file	0.35		
roll axis wheel		0.0007	0.0139
pitch axis wheel		0.0014	0.0102
yaw axis wheel		0.0018	0.0068
discrete MATLAB func.	1.75		
roll axis wheel		8.6×10^4	2.9×10^5
pitch axis wheel		233.480	394.034
yaw axis wheel		233.883	434.861

Also, a pure discrete-time simulation of the wheel assembly (i.e., no ANN) , sampled at 1.024 seconds, was written for comparison with the ANN-based model simulations. This simulation was also written as a MATLAB function, and executed in MATLAB v5.0.

The results in Table 2 show that although the pure discrete wheel model simulation still executed at a faster rate (although it should be noted that its execution time did increase slightly, where as the ANN-based model simulation execution times were about the same as in the previous case), its outputs were physically meaningless. Figure 8. shows the roll axis wheel speed output time histories for this friction case, from the 'true model' Simulink simulation (top), from the ANN-based model MEX file simulation (middle), and the pure discrete-time model simulation (bottom). Clearly, the ANN-based simulation results matched the 'true model' results much better, while the pure discrete-time model simulation results were physically meaningless for the given sampling period of 1.024 seconds. The ANN-based wheel model simulation gave wheel output results which were very comparable to the 'true model' results, while executing about 180 times faster, which was a very sizable speed up.

It should be noted that in those cases where the nonlinearity and/or rapid dynamics (relative to the discrete step size) in the system are not significant, then pure discrete-time model simulation performance was found to be comparable to the ANN-based model simulations. For example, in other tests using the

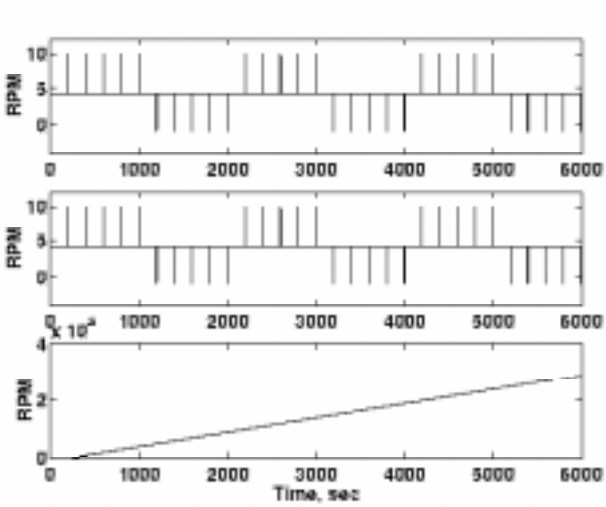

Figure 8. Roll axis wheel speed simulation results.

same reaction wheel model, with the wheel friction being more linear (i.e., the coefficient associated with the quadratic term in the viscous friction equation Eq. (2) reduced in magnitude by a factor of 1000), the differences in the ANN-based model simulation results and the pure discrete-time model simulations results were practically negligible. The same held true for simulations with torque command input profiles, such as low frequency pure sine waves, which caused only slow varying time response outputs from the wheel model. Therefore, the real advantage of ANNs is truly seen when the dynamic model that is to be approximated is commandingly nonlinear.

Conclusions

This paper presented a specific application of ANNs for rapid and efficient dynamics and control analysis of flexible systems. Specifically, feedforward neural networks were designed to approximate the dynamics of components (over prescribed input ranges), for use in simulations as a means to speed up the overall time response analysis process. To capture the recursive nature of dynamic components with artificial neural networks, recurrent networks, which used state feedback with the appropriate number of time delays, as inputs to the networks, were employed. Once properly trained, neural networks gave very good approximations to nonlinear dynamic components at a fraction of the cost of full nonlinear dynamic integration, and by their judicious use, have paved a way for a potential speed up in the overall analysis process. To illustrate this potential speed up, an existing simulation model of a spacecraft reaction wheel system was used, first conventionally and then with an ANN-based model in place. Simulation results indicated that, at least in the presence of significant model nonlinearity or command input profiles which cause fast varying responses, the ANN-based model gave accurate answers, with computational speed-ups up to a factor of 180.

References

[1]S. Hayden, *Neural Networks: A Comprehensive Foundation*, Macmillan College Publishing Co., New York, 1994.

[2] D.E. Rumelhart and J.L. McClelland, *Parallel Distributed Processing, Vol.1*, MIT Press, Cambridge, MA, 1986.

[3]M.T. Hagan and M.B. Menhaj, "Training Feedfoward Networks with the Marquardt Algorithm", *IEEE Transactions on Neural Networks*, Vol. 5, No. 6, November 1994, pp. 989-993.

[4] K.S. Narenda, "Adaptive Control of Dynamical Systems Using Neural Networks", *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, ed. by D.A. White and D.A. Sofge, Van Nostrand Reinhold, New York, 1992, pp. 141-183.

[5] K. Funahashi, "On the Approximate Realization of Continuous Mappings by Neural Networks", *Neural Networks*, Vol. 2, 1989, pp. 183-192.

[6] A.R. Gallant and H. White, "There Exists a Neural Network That Does Not Make Avoidable Mistakes", *Proceedings of the IEEE 2nd International Conference on Neural Networks*, 1988, pp. 657-664.

[7] M.I. Elmasry (ed.), *VLSI Artificial Neural Networks Engineering*, Kluwer Academic Publishers, Norwell, MA, 1994.

[8]K. Wawryn and B. Streszewski, "Low Power VLSI Neuron Cells for Artificial Neural Networks", *Proceedings of the 1996 IEEE International Symposium on Circuits and Systems*, 1996, pp. 372-375.