IMPLEMENTATION OF A NEURAL NET TRACKING CONTROLLER FOR A SINGLE FLEXIBLE LINK: COMPARISON WITH PD AND PID CONTROLLERS

The members of the Committee approve the masters thesis of Luis Benigno Gutiérrez

Frank L. Lewis
Supervising Professor

Kai Liu

Michael T. Manry

To the memory of my father,

with his example he inspired

me to reach all my goals

IMPLEMENTATION OF A NEURAL NET TRACKING CONTROLLER FOR A SINGLE FLEXIBLE LINK: COMPARISON WITH PD AND PID CONTROLLERS

by

LUIS BENIGNO GUTIERREZ

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 1996

ACKNOWLEDMENTS

I would like to express my appreciation to Dr. Frank L. Lewis who gave me the opportunity to increase my knowledge and have contact with real research at the Automation and Robotics Research Institute. This thesis could not have been written without his guidance and support.

I would also like to thank Andy Lowe. He implemented the PD controller for the flexible link, which was the base for the implementation of the neural net controller. His help was very valuable to solve some of the problems presented in the implementation.

Finally, I would like to thank Dr. Kai Liu and Dr. Michael T. Manry for being part of my thesis committee, and all the professors who guided me during my stay at UT Arlington.

July 24, 1996

ABSTRACT

IMPLEMENTATION OF A NEURAL NET TRACKING CONTROLLER FOR A SINGLE FLEXIBLE LINK: COMPARISON WITH PD AND PID CONTROLLERS

Publication No.________

Luis Benigno Gutiérrez, M.S.

The University of Texas at Arlington, 1996

Supervising Professor: Frank L. Lewis

The objective of this thesis is to show the results of the practical implementation of a neural network tracking controller on a single flexible link and compare its performance to that of PD and PID standard controllers. The NN controller is composed of an outer PD tracking loop, a singular perturbation inner loop for stabilization of the fast flexible mode dynamics, and a neural network inner loop used to feedback linearize the slow pointing dynamics. No off-line training or learning is needed for the NN. It is shown that the tracking performance of the NN controller is far better than that of the PD or PID standard controllers. An extra friction term was added in the tests to demonstrate the ability of the NN to learn unmodeled nonlinear dynamics.

TABLE OF CONTENTS

ACKNOWLEDMENTS iv

ABSTRACT v

LIST OF FIGURES viii

Chapter

1. INTRODUCTION 1

2. BACKGROUND 4

 2.1. Control of Flexible Link Robots 4

 2.2. Applications of Neural Networks for Control 6

3. DYNAMICS OF A FLEXIBLE LINK ROBOT ARM 15

4. NEURAL NET CONTROL OF FLEXIBLE LINK ROBOT ARMS 21

 4.1. Singular Perturbation Approach 21

 4.2. Neural Net Control of the Rigid Dynamics 24

 4.3. Overall Control Structure 27

 4.4. Simulation 28

5. IMPLEMENTATION OF THE CONTROLLER IN THE FLEXIBLE-LINK TEST-BED 31

 5.1. Description of the Implementation 31

 5.2. Experimental Results in the Flexible Link Test-bed 35

  5.2.1. PD control 35

  5.2.2. PID Control 36

  5.2.3. NN+PD Control 38

Chapter

5.3. Comparison between different approaches 40

6. CONCLUSIONS 45

Appendix

A. C CODE OF THE FUNCTIONS MORE RELEVANT IN THE INPLEMENTATION OF THE NN CONTROLLER 46

REFERENCES 72

LIST OF FIGURES

Figure Page

1. Representation of a neuron 7
2. Sigmoid activation function 8
3. One-layer neural network 9
4. Two-layer neural network 11
5. Coordinate frames for the single flexible link 17
6. Overall control structure of the neural network controller for a flexible link robot 28
7. Simulation of the neural network controller with a single flexible link 30
8. Actual neural network controller implemented at ARRI's flexible link test-bed 32
9. Flexible link controller implementation at ARRI 33
10. Block diagram of neural network controller implementation at ARRI's flexible link test-bed 34
11. Performance of the PD control. (a) Without additional friction. (b) With additional friction. 37
12. Performance of the PID control. (a) Without additional friction. (b) With additional friction. 39
13. Performance of the PD+NN control before learning. (a) Without additional friction. (b) With additional friction. 41
14. Performance of the PD+NN control after learning. (a) Without additional friction. (b) With additional friction. 42

Figure Page

15. Step response of the controllers. (a) PD control. (b) PD control with extra friction. (c) PID control. (d) PID control with extra friction. (e) PD+NN control. (f) PD+NN control with extra friction. 44

CHAPTER 1

INTRODUCTION

In recent literature there have been many neural network controllers proposed for robot arms or other nonlinear systems [4],[25],[26],[29],[30],[31],[32]. The performance of these neural net controllers on actual systems has been open to question, despite the fact that several of these references provide stability proofs. In this paper we implement the neural net controller derived in [40] on an actual single-flexible-link robot arm which could emulate, for instance, a tank gun barrel in DoD applications. It is found that the NN controller far outperforms standard PD and PID controllers, even for the single-link arm which is basically linear except for nonlinear friction effects.

The control of flexible link robot arms belongs to a class of problems characterized by having reduced control effectiveness and an additional unstable zero dynamics. Some other problems in this category are large-scale space structures, overhead gantry cranes, and other industrial processes. The requirement of controllers with faster response and higher accuracy introduces a challenge that the researchers have faced in different ways.

Several researchers [24],[37] have observed that the approximate flexible-link robot arm dynamics is input-output feedback linearizable but the zero dynamics is not asymptotically stable when the tip position is taken as the output. To control the arm, a modified output was defined to yield stable zero dynamics. However, this output does not correspond to practical tracking objectives except in the set-point command case. In [34],

[36] input-output feedback linearization and a singular perturbation correction term [15] to stabilize the internal dynamics was used to control a multi-link flexible arm. Finally, in [22] a Lyapunov approach is used to stabilize a component of the tracking error, but not the tracking error in its entirety.

There are different control techniques for rigid robot arms available in the literature. These techniques require an exact knowledge about the nonlinear terms (computed torque), knowledge of bounds on uncertainties (robust control), or knowledge of a nonlinear regression matrix of robot functions (adaptive control) [19]. In practice it is very difficult to have such a priori knowledge of the arm dynamics, especially in the presence of frictional terms which may not have a known dynamical form.

To overcome these limitations a neural net tracking controller for a rigid link robot arm has been devised in [17],[20]. In this scheme there is an outer PD tracking loop with the neural network used in a feedback linearization inner loop. The weight training rules include an e-modification term [27] and a term corresponding to a second-order term. Using a Lyapunov approach it is shown that these training rules guarantee tracking performance and bounded weights even though there do not exist ideal weights such that the neural net perfectly reconstructs the nonlinear robot function.

In [40], a tracking controller for a flexible-arm is designed using singular perturbation plus a NN feedback linearization inner loop. There, a modified output for tracking is defined that does correspond to practical tracking requirements. The structure of that controller includes an outer PD tracking loop, a singular perturbation inner loop for stabilization of the fast dynamics, and a neural network inner loop used to feedback linearize the rigid dynamics. Applying singular perturbation theory it is shown that after stabilizing the fast dynamics, the slow dynamics can be controlled using the same approach used in [17] and [20]. This approach avoids the requirement of the knowledge of friction, gravity and coriolis/centripetal terms, or any regression matrix. In contrast to other NN controllers in the literature, there is no off-line learning phase, the NN weights are easy to initialize without known 'stabilizing initial weights' (the weights are initialized at zero), and the controller guarantees boundedness of the tracking error and control signal.

In this thesis, some practical implementation results for a single flexible link for the controller designed in [40] are presented. Despite the fact that the dynamics of a single flexible link is linear, an extra friction term was added in the implementation to show the capability of the NN controller to compensate for nonlinearities in the model by learning. A comparison with the performance of standard PD and PID controllers is performed to show the superior tracking performance of the NN controller.

CHAPTER 2

BACKGROUND

2.1. Control of Flexible Link Robots

Some of the approaches that have been used before to control flexible link robots are discussed here.

In [34], the singular perturbation approach is proposed to control a flexible manipulator. The control action is composed of a slow component designed for the slow subsystem, and a fast component designed to stabilize the fast subsystem around the equilibrium trajectory set up by the slow subsystem under the effect of the slow control. This approach is followed in [36] using a computed torque technique to control the slow subsystem. These methods represent the dynamics of the flexible manipulators using a Lagranian-assumed modes formulation.

In [33] the singular perturbation method was used for a two-link flexible manipulator. In contrast to the methods mentioned before, the dynamics of the flexible manipulator was represented by a distributed parameter model. In this case, the slow control is a decentralized torque control which approximately linearizes the reduced order model and adds a PI control for trajectory tracking. The fast control is a distributed actuator (a piezoelectrically active film) used to dampen the flexural vibrations.

All of the following control strategies are based in the Lagranian-assumed modes formulation for the dynamics of the flexible manipulators.

In [24] the trajectory tracking control of the tip position of a two-link elastic manipulator based on nonlinear inversion and linear stabilization was considered. The outputs were chosen as the sum of the joint angle and tip elastic deformation times a constant factor for each link. This way, the problem of unstable zero dynamics in the closed-loop system when the exact tip position is taken as output was avoided. A linear stabilizer was designed for the zero dynamics. The effectiveness of this strategy was checked through simulations but no formal proof of the closed loop stability of the system was given.

In [37] the transfer function of a single flexible link was obtained using the Lagranian-assumed-modes approach. It was shown that the transfer function becomes ill defined when the number of modes retained in the model is increased if the tip position is taken as output. Taking the tip position minus the elastic deformation as the output, the transfer function becomes well-defined with a relative degree of two. An H_2-optimal controller was found and simulations were performed to illustrate the advantages of the proposed transfer function.

In [39] the control of a class of manipulators with a single flexible link was addressed. The state space equations of the manipulator were transformed into an equivalent set of equations that are almost linear. The controller used a nonlinear state feedback which was designed based only in the linear part of the transformed equations, and was combined with an observer. This scheme was shown to be input-output stable in a local sense.

In [28] a controller for a two-link flexible arm was designed based on variable structure system (VSS) theory. The joint angles were controlled using a variable structure control (VSC) law which included the integral of the tracking error, and the elastic oscillations of the links were stabilized using the pole assignment technique. It was shown that the closed loop system including the sliding mode controller is stable.

In [23] a feedback linearization/fuzzy logic controller for a flexible link manipulator was designed. In this scheme a reduced order computed torque (ROCT) control was first used to linearize the whole system to a Newton's law like system, then a fuzzy logic controller consisting of 33 condition-action rules was used to command the rigid modes to track the desired trajectories while maintaining the residual vibrations as small as possible.

2.2. Applications of Neural Networks for Control

Inspired in the model of the human brain, researchers have developed the so called artificial neural networks [11],[18]. A neural network (NN) is a system composed of the interconnection of many simple nonlinear elements called neurons. Each neuron (sometimes called perceptron) obtains its output from the application of a nonlinear function (called activation function) to a linear combination of its inputs plus a threshold (figure 1). Therefore, for a single neuron we can write

$$y = \sigma (v^T x) \tag{2-1}$$

with

$$x = [1 \quad x_1 \quad x_2 \quad \cdots \quad x_n]^T, \; v = [v_0 \quad v_1 \quad v_2 \quad \cdots \quad v_n]^T, \tag{2-2}$$

where y is the output, $x_1, x_2, \ldots, x_n$ are the inputs, v_0 is the threshold , $v_1, v_2, \ldots, v_n$ are the weights, and $\sigma(\bullet)$ is the activation function.

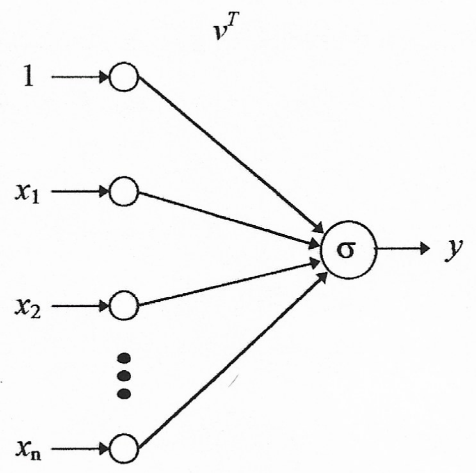


Fig. 1. Representation of a neuron.

A common choice of the activation functions are the sigmoid functions. These are monotonically non-decreasing functions taking on bounded values at $-\infty$ and $+\infty$. For the use of the backpropagation training algorithm it is required that the sigmoid function be differentiable. The activation functions used in this thesis are the sigmoid functions defined by

$$\sigma(z) = \frac{1}{1+e^{-\alpha z}}, \tag{2-3}$$

where α is a parameter that determines the slope of the sigmoid function around zero. See figure 2.

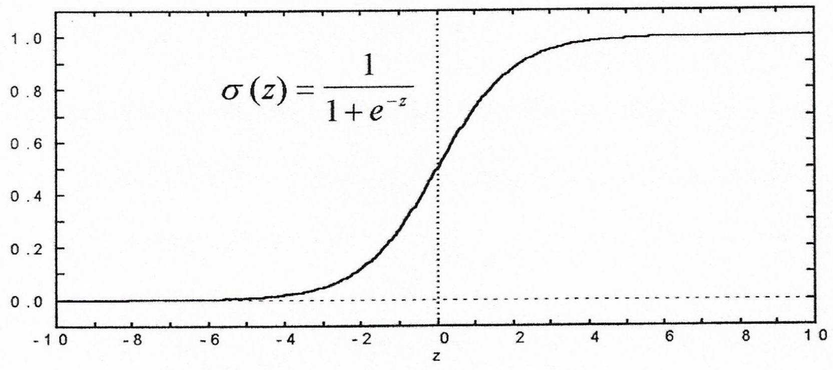


Fig. 2. Sigmoid activation function.

Having several neurons with the same inputs we can construct a one-layer NN (figure 3) which is defined by

$$y = \sigma(V^T x), \tag{2-4}$$

with

$$x = [1 \quad x_1 \quad x_2 \quad \cdots \quad x_n]^T, \; y = [y_1 \quad y_2 \quad \cdots \quad y_l]^T, \tag{2-5}$$

where y is the output vector (with the l outputs $y_1, y_2, \ldots, y_l$), $x_1, x_2, \ldots, x_n$ are the n inputs, V is the weights matrix given by

$$V^T = \begin{bmatrix} v_{10} & v_{11} & v_{12} & \cdots & v_{1n} \\ v_{20} & v_{21} & v_{22} & \cdots & v_{2n} \\ \vdots & \vdots & & & \vdots \\ v_{l0} & v_{l1} & v_{l2} & \cdots & v_{ln} \end{bmatrix}, \tag{2-6}$$

where the first column contains the thresholds, and the other columns are the weights, and $\sigma(\bullet)$ is the activation function vector given by

$$\sigma(z)=\begin{bmatrix}\sigma_1(z_1)\\ \sigma_2(z_2)\\ \vdots\\ \sigma_l(z_l)\end{bmatrix} \quad \text{for} \quad z=\begin{bmatrix}z_1\\ z_2\\ \vdots\\ z_l\end{bmatrix} \quad \text{and} \quad \sigma_i(z)=\frac{1}{1+e^{-\alpha z_i}}. \tag{2-7}$$

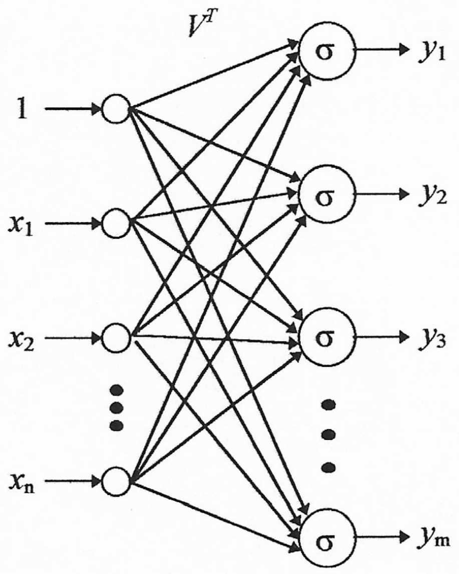


Fig. 3. One-layer neural network.

It was found by the researchers that the one-layer NN could not approximate many simple functions. For example, AND, OR, and NOT logic operators can be constructed using one-layer NN, but not the XOR logic operator [18]. Later it was found that a two-layer NN could approximate a XOR logic operator and more complicated functions, this motivated a lot of research in the multilayer NN (sometimes called multilayer perceptron). A two-layer NN (figure 4) is defined by

$$y = \sigma(W^T \sigma(V^T x)) \tag{2-8}$$

with

$$x = [1 \quad x_1 \quad x_2 \quad \cdots \quad x_n]^T,\ y = [y_1 \quad y_2 \quad \cdots \quad y_m]^T, \tag{2-9}$$

where y is the output vector (with the m outputs $y_1, y_2, \ldots, y_m$), $x_1, x_2, \ldots, x_n$ are the n inputs, V is the input weights matrix given by

$$V^T = \begin{bmatrix} v_{10} & v_{11} & v_{12} & \cdots & v_{1n} \\ v_{20} & v_{21} & v_{22} & \cdots & v_{2n} \\ & \vdots & & & \vdots \\ v_{l0} & v_{l1} & v_{l2} & \cdots & v_{ln} \end{bmatrix}, \tag{2-10}$$

with l number of "hidden" neurons and W is the output weights matrix given by

$$W^T = \begin{bmatrix} w_{10} & w_{11} & w_{12} & \cdots & w_{1l} \\ w_{20} & w_{21} & w_{22} & \cdots & w_{2l} \\ & \vdots & & & \vdots \\ w_{m0} & w_{m1} & w_{m2} & \cdots & w_{ml} \end{bmatrix}, \tag{2-11}$$

where the first column contains the thresholds, and the other columns are the weights, and $\sigma(\bullet)$ is the activation function vector with the appropriate number of components for each case. The first layer of neurons is referred as the hidden layer since its outputs are not seen at the outputs of the NN. The activation function vector includes a 1 in the first component for the hidden layer (that is, in the computation of $z = \sigma(V^T x)$, where z is the vector of the outputs of the neurons in the first layer). This makes easier to include the thresholds in the formulation of (2-8).

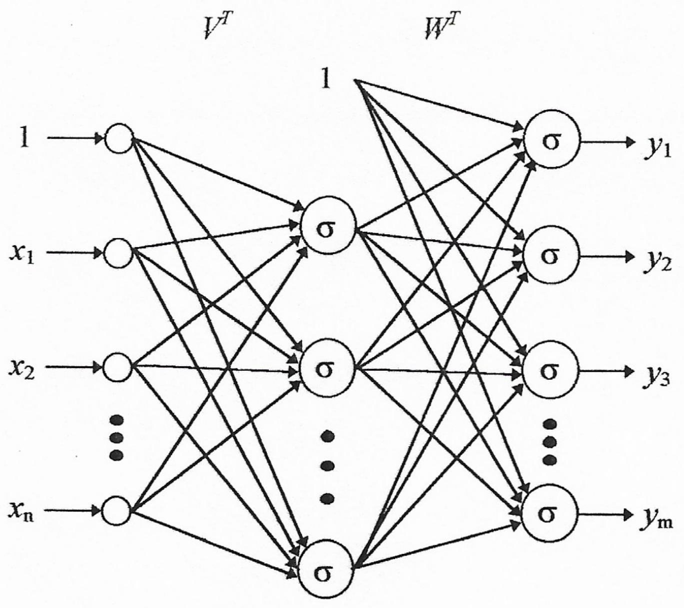


Fig. 4. Two-layer neural network.

Neural networks like the ones defined in (2-4) or (2-8) are static (they contain no integrators or time delays). There are many kinds of dynamic NN in which some of the signals are fed back through integrators or delay units. One of the most familiar dynamic NN is the Hopfield NN.

In most of the applications to Digital Signal Processing (DSP), the classification, association, and pattern recognition properties of neural networks are used. In such applications the main purpose of the neural networks is to distinguish between different inputs associating them with the closest of a set of exemplar patterns.

A NN like the one defined in (2-8) has an important property called "universal approximation property." In closed loop control applications the universal approximation property of multilayer NN is of great importance. This property is stated in the following theorem.

Theorem: Universal Approximation Property of the NN

Let $f(x): \Re^n \rightarrow \Re^m$ be a smooth function. Then , given a compact set $S \in \Re^n$ and a positive number ε_N , there exists a two-layer NN such that

$$f(x) = W^T \sigma(V^T x) + \varepsilon \tag{2-12}$$

with $\|\varepsilon\| < \varepsilon_N$ for all $x \in S$, for some (sufficiently large) number L of hidden layer neurons. ε is generally a function of x and is called the NN function approximation error. ε decreases as L increases. □

A key issue in the success of the applications of neural networks is the training algorithm used to select the weights. The training of the NN consists in updating the weights using certain rules which could be continuous (differential equations) or discrete (difference equations). This training is what gives the NN learning capabilities. There are three categories for the learning schemes: supervised learning, unsupervised learning, and reinforcement learning. When the information needed to train the NN is known a priori (the pairs of inputs and desired outputs are known) supervised learning is used. In this case there is a "teacher" which applies the inputs and change the weights based on the error respect to the desired outputs. In unsupervised learning there is no teacher with global information to train the NN, so the training is performed based on local data which is examined and organized according to emergent collective properties. Finally, in reinforcement learning, the weights associated with a particular neuron are changed in proportion to some global reinforcement signal.

In the operation of a NN there are two phases: the learning phase, when the NN is trained, and the operational phase, when the NN performs its design function. A NN is said to use off-line learning if the learning phase is carried out first and then the operational phase occurs with the weights fixed at the values obtained in the learning phase. This kind of learning is used in most DSP applications and some open loop control applications. On the other hand, in on-line learning the learning phase and the operational phase occur at the same time, so the NN learns while it is operating in the actual application. This kind of learning is common in closed loop control applications. For instance, the controller discussed in this thesis falls into this category.

Here, the backpropagation training algorithm is briefly discussed, since it is the one that concerns more with the application presented in this thesis. This algorithm is a gradient descent algorithm for multilayer neural networks. Considering the two-layer NN from equation (2-8) the algorithm can be divided in three stages for each iteration.

First, the NN is presented with an input pattern X and its output is computed using

$$z = \sigma(W^T x), \tag{2-13}$$

and

$$y = \sigma(V^T z). \tag{2-14}$$

Second, given the desired output Y for the input X, a backward recursion is used to compute the backpropagated errors

$$\begin{aligned} e &= Y - y \\ \delta_i^2 &= y_i(1-y_i)e_i \ ; \ i = 1,2,\ldots,m \\ \delta_i^1 &= z_l(1-z_l)\sum_{\mathrm{i=1}}^{\mathrm{m}} w_{il}\delta_i^2 \ ; \ l = 1,2,\ldots,L \end{aligned} \quad . \tag{2-15}$$

Finally, the NN weights and thresholds are updated by

$$\begin{aligned} W &= W + \eta\delta^2 z^T \\ V &= V + \eta\delta^1 X^T \end{aligned} , \tag{2-16}$$

where

$$\delta^1 = \begin{bmatrix} \delta_1^1 \\ \delta_2^1 \\ \vdots \\ \delta_L^1 \end{bmatrix} \text{ and } \delta^2 = \begin{bmatrix} \delta_1^2 \\ \delta_2^2 \\ \vdots \\ \delta_m^2 \end{bmatrix} . \tag{2-17}$$

and η is the learning rate, which in some cases is varied adaptively. The iterations should be repeated until the output error has become sufficiently small.

CHAPTER 3

DYNAMICS OF A FLEXIBLE LINK ROBOT ARM

In [5],[6],[9],[10],[21] it is shown that the dynamics of any multi-link Flexible Link Robot can be represented by

$$M(q)\ddot{q} + D(q,\dot{q})\dot{q} + Kq + F(q,\dot{q}) + G(q) = B(q)u\,, \qquad (3\text{-}1)$$

with

$$q = \begin{bmatrix} q_r \\ q_f \end{bmatrix}$$

where q_r is the vector of rigid modes (generalized joint coordinates) and q_f is the vector of flexible modes (the amplitudes of the flexible modes). *M(q)* represents the inertia matrix, $D(q,\dot{q})$ is the Coriolis and centrifugal matrix, *K* is the stiffness matrix, $F(q,\dot{q})$ is the friction matrix, *G(q)* is the gravity matrix, *B(q)* is an input matrix dependent on the boundary conditions selected in the assumed mode shapes method, and *u* includes the control torques applied to each joint.

The model (3-1) follows the same properties of any standard rigid link robot [21]. That is *M(q)* is positive definite and upper and lower bounded, $D(q,\dot{q})$ is bounded by $d_b(q)\|\dot{q}\|$, and $D(q,\dot{q})$ can be chosen such that $\dot{M}(q) - 2D(q,\dot{q})$ is skew-symmetric [21].

There are different ways to obtain the model of a flexible arm [12]: using the Lagranian method or the Euler-Newton method, using a modal expansion (assumed modes method) or a finite elements method, using the Hamilton's principle, and using nonstructural modeling.

Here, the dynamic model of a single flexible link is obtained using the Lagranian-assumed modes method as shown in [10],[37]. It is assumed that the height of the beam is much greater than the width, so the beam is constrained to move in the horizontal direction. The effects of shear deformation and rotary inertia are neglected and the deflections of the beam are assumed to be small. The beam has a moment of inertia J_b, a mass density ρ, a cross section A, and a length h. The elastic deformation of the beam at a distance x from the hub is $w(x,t)$. The beam is assumed to be fixed to the shaft of the motor which produces a torque u. A coordinate frame x'-y' is attached to the flexible link at the point where the beam is attached to the motor hub. This frame rotates with the beam respect to the base coordinate frame x_0-y_0 such that the slope of the beam at x=0 is always zero respect to x'-y' (see figure 5). The rotation angle of x'-y' respect to x_0-y_0 is the rigid mode q_r.

The elastic deformation is modeled by the Bernoully-Euler equation ([7],[8])

$$-\frac{\partial^2 w(x,t)}{\partial^2 t} = \frac{EI}{\rho A}\frac{\partial^4 w(x,t)}{\partial^4 x}\ , \tag{3-2}$$

subject to the clamped-free boundary conditions

$$w(0,t)=0\ ,\ w'(0,t)=0\ ,\ w''(h,t)=0\ ,\ w'''(h,t)=0. \tag{3-3}$$

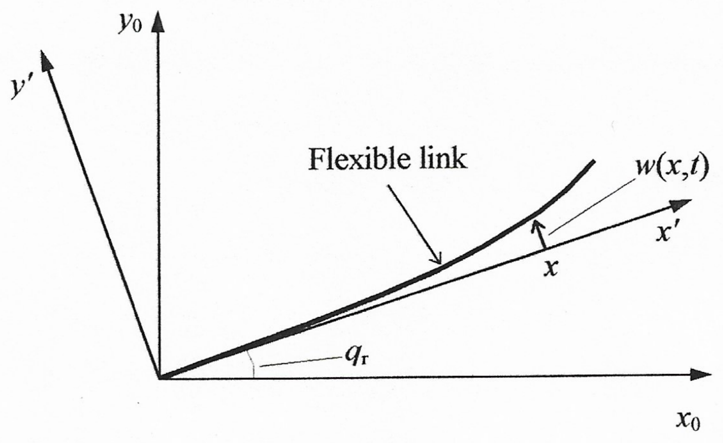


Fig. 5. Coordinate frames for the single flexible link.

In the assumed-modes method the deformation *w*(*x*,*t*) is expanded in a series of the form

$$w(x,t) = \sum_{i=1}^{n} q_{fi}(t)\phi_i(x) \tag{3-4}$$

where $q_{fi}(t)$ is the amplitude of the flexible mode *i* which is only function of *t*, $\phi_i(x)$ is the eigenfunction for mode *i* which is only function of *x*, and *n* is the number of modes retained in the model.

Equation (3-2) is solved using separation of variables obtaining the following ordinary diferential equations for $q_{fi}(t)$ and $\phi_i(x)$

$$\frac{d^2 q_{fi}(t)}{dt^2} + \omega^2 q_{fi}(t) = 0 \tag{3-5}$$

$$\frac{d^4\phi_i(x)}{dt^4} - \beta^4\phi_i(x) = 0 \tag{3-6}$$

with

$$\beta^4 = \frac{\rho A\omega^2}{EI} \tag{3-7}$$

The solutions for these equations, under the boundary conditions of (3-3), are

$$q_{fi}(t) = A_i\cos(\omega_i t + \alpha_i) \tag{3-8}$$

and

$$\phi_i(x) = c_i\left[(\sin\beta_i x - \sinh\beta_i x) - \frac{(\sin\beta_i h + \sinh\beta_i h)}{(\cos\beta_i h + \cosh\beta_i h)}(\cos\beta_i x - \cosh\beta_i x)\right] \tag{3-9}$$

where the β_i are the solutions to

$$\cos\beta_i h\cosh\beta_i h = -1 \tag{3-10}$$

and c_i are the constants that normalize the eigenfunctions so that

$$\int_0^h \phi_i(x)\,dx = 1 \tag{3-11}$$

To derive the dynamic model of the link, the Lagranian $L = K - V$ is first computed. The position of a point on the beam at a distance x from the hub is given by

$$P(x) = \begin{bmatrix} x\cos q_r - w\sin q_r \\ x\sin q_r + w\cos q_r \end{bmatrix}, \tag{3-12}$$

hence

$$\dot{P}^T\dot{P} = x^2\dot{q}_r^2 + \dot{w}^2 + 2\dot{w}x\dot{q}_r + w^2\dot{q}_r^2, \tag{3-13}$$

so the kinetic energy is

$$\begin{aligned} K &= \frac{1}{2} J_h \dot{q}_r^2 + \frac{1}{2}\int_0^h \dot{P}^T \dot{P}\, dm \\ &= \frac{1}{2} J_h \dot{q}_r^2 + \frac{1}{2}\int_0^h x^2\, dm\, \dot{q}_r^2 + \frac{1}{2}\int_0^h \dot{w}^2\, dm + \int_0^h \dot{w} x \dot{q}_r\, dm + \frac{1}{2}\int_0^h w^2\, dm\, \dot{q}_r^2 \end{aligned}, \quad (3\text{-}14)$$

where J_h is the hub inertia. The potential energy is

$$V = \frac{1}{2}\int_0^h EI \left(\frac{\partial^2 w}{\partial x^2} \right)^2 dx\,. \quad (3\text{-}15)$$

Assuming that w is very small, the expression

$$\int_0^h (x^2 + w^2)\, dm \quad (3\text{-}16)$$

can be approximated by

$$J_b = \int_0^h x^2\, dm\,. \quad (3\text{-}17)$$

The Lagranian $L = K - V$ can be found to be

$$L = \frac{1}{2}(J_h + J_b)\dot{q}_r^2 + \frac{1}{2}\sum_{i=1}^{n} \dot{q}_{fi}^2 + \dot{q}_r \sum_{i=1}^{n} \dot{q}_{fi} \int_0^{\mathrm{h}} \phi_{\,i} x\, dm - \frac{1}{2}\sum_{i=1}^{n} q_{fi}^2 \omega_{\mathrm{i}}^2 \quad (3\text{-}18)$$

Using q_r and $q_{fi}(t)$ as the generalized coordinates and applying the Euler-Lagrange equations, the equations of motion are obtained

$$\begin{aligned} &\frac{d}{dt}\frac{\partial L}{\partial \dot{q}_r} - \frac{\partial L}{\partial q_r} = u \\ &\frac{d}{dt}\frac{\partial L}{\partial \dot{q}_{fi}} - \frac{\partial L}{\partial q_{fi}} = 0\ , \quad i = 1,2,\ldots,n \end{aligned} \quad (3\text{-}19)$$

Substituting (3-18) into (3-19) gives the model of the flexible link

$$M(q)\ddot{q} + D(q,\dot{q})\dot{q} + Kq = B(q)u \quad (3\text{-}20)$$

where the inertia matrix is

$$M(q) = \begin{bmatrix} J_h + J_b & \gamma_1 & \gamma_2 & & \gamma_n \\ \gamma_1 & m & 0 & \cdots & 0 \\ \gamma_2 & 0 & m & & 0 \\ & \vdots & & & \vdots \\ \gamma_n & 0 & 0 & \cdots & m \end{bmatrix} \tag{3-21}$$

with m the mass of the beam and

$$\gamma_i = \int_0^h \phi_i x \, dm \ , \quad i = 1,2,\ldots,n \ , \tag{3-22}$$

the stiffness matrix is

$$K = \begin{bmatrix} 0 & 0 & 0 & & 0 \\ 0 & \omega_1^2 & 0 & \cdots & 0 \\ 0 & 0 & \omega_2^2 & & 0 \\ & \vdots & & & \vdots \\ 0 & 0 & 0 & \cdots & \omega_n^2 \end{bmatrix}, \tag{3-23}$$

and

$$B(q) = \begin{bmatrix} 1/m \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{3-24}$$

The matrix $D(q,\dot{q})$ includes the damping factors ζ_i for the flexible modes q_{fi}

$$D(q,\dot{q}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 2\zeta_1\omega_1 & 0 & 0 \\ 0 & 0 & 2\zeta_2\omega_2 & 0 \\ & & & \\ 0 & 0 & 0 & 2\zeta_n\omega_n \end{bmatrix}. \tag{3-25}$$

CHAPTER 4

NEURAL NET CONTROL OF FLEXIBLE LINK ROBOT ARMS

4.1. Singular Perturbation Approach

The singular perturbation approach basically consists in breaking the dynamics of the system in two parts, each of them in a separate time scale [14],[15],[16]. In this case the slow dynamics corresponds to the rigid modes q_r and the fast dynamics corresponds to the flexible modes q_f. In order to apply singular perturbation, (3-1) can be split as in [34],[36]

$$\begin{aligned} \ddot{q}_r &= -D^1_{rr}\dot{q}_r - D^1_{rf}\dot{q}_f - K^1_{rf}q_f - F^1_r - G^1_r + B^1_r u \\ \ddot{q}_f &= -D^1_{fr}\dot{q}_r - D^1_{ff}\dot{q}_f - K^1_{ff}q_f - F^1_f - G^1_f + B^1_f u \end{aligned} \quad . \tag{4-1}$$

Now introduce the scale factor ε and define

$$\varepsilon^2\xi = q_f \, , \tag{4-2}$$

where $1/\varepsilon^2$ is the smallest stiffness in K^1_{ff}. Define

$$\widetilde{K}_{ff} \equiv \varepsilon^2 K^1_{ff} \, . \tag{4-3}$$

Then

$$\begin{aligned} \ddot{q}_r &= -D^1_{rr}(\theta)\dot{q}_r - D^1_{rf}(\theta)\varepsilon^2\dot{\xi} - H_{rf}(1/\varepsilon^2)\widetilde{K}_{ff}(q_r,\varepsilon^2\xi)\varepsilon^2\xi - F^1_r(q_r,\varepsilon^2\xi) - G^1_r(q_r,\varepsilon^2\xi) + B^1_r(q_r,\varepsilon^2\xi)u \\ \varepsilon^2\ddot{\xi} &= -D^1_{fr}(\theta)\dot{q}_r - D^1_{ff}(\theta)\varepsilon^2\dot{\xi} - (1/\varepsilon^2)\widetilde{K}_{ff}(q_r,\varepsilon^2\xi)\varepsilon^2\xi - F^1_f(q_r,\varepsilon^2\xi) - G^1_f(q_r,\varepsilon^2\xi) + B^1_f(q_r,\varepsilon^2\xi)u \end{aligned} \tag{4-4}$$

where $\theta = (q_r, \dot{q}_r, \varepsilon^2\xi, \varepsilon^2\dot{\xi})$. Here is considered the case in which the stiffness of the links is sufficiently large so ε is sufficiently small. The control objective is that $q_r(t)$ should track $q_d(t)$, a prescribed trajectory. For that purpose define the control

$$u = \overline{u} + u_F \tag{4-5}$$

where $\overline{u}$ is the slow component and u_F is the fast component.

To obtain the equations for the slow dynamics, set $\varepsilon = 0$ in (4-4) to obtain

$$\ddot{\overline{q}}_r = -\overline{D}_{rr}^1\dot{\overline{q}}_r - \overline{H}_{rf}\widetilde{K}_{ff}\overline{\xi} - \overline{F}_r^1 - \overline{G}_r^1 + \overline{B}_r^1\overline{u} \tag{4-6}$$

and the algebraic slow manifold equation

$$0 = -\overline{D}_{fr}^1\dot{\overline{q}}_r - \overline{H}_{ff}\widetilde{K}_{ff}\overline{\xi} - \overline{F}_f^1 - \overline{G}_f^1 + \overline{B}_f^1\overline{u}\,, \tag{4-7}$$

which is solved for the slow variables

$$\overline{\xi} = \widetilde{K}_{ff}^{-1}\overline{H}_{ff}^{-1}(-\overline{D}_{fr}^1\dot{\overline{q}}_r - \overline{F}_f^1 - \overline{G}_f^1 + \overline{B}_f^1\overline{u})\,. \tag{4-8}$$

Substituting (4-8) in (4-6), we get

$$\ddot{\overline{q}}_r = \overline{M}_{rr}^{-1}(-\overline{D}_{rr}\dot{\overline{q}}_r - \overline{F}_r - \overline{G}_r + \overline{u})\,. \tag{4-9}$$

For the fast subsystem define the states

$$\begin{aligned} \varsigma_1 &\equiv \xi - \overline{\xi} \\ \varsigma_2 &\equiv \varepsilon\dot{\xi} \end{aligned} \tag{4-10}$$

with a time scale $\tau = t/\varepsilon$ resulting in

$$\begin{aligned} \frac{d\varsigma_1}{d\tau} &= \varsigma_2 \\ \frac{d\varsigma_2}{d\tau} &= -\overline{D}_{fr}^1\dot{\overline{q}}_r - D_{ff}^1\varepsilon\varsigma_2 - \overline{H}_{ff}\widetilde{K}_{ff}(\varsigma_1 + \overline{\xi}) - \overline{F}_f^1 - \overline{G}_f^1 + \overline{B}_f^1(\overline{u} + u_F) \end{aligned} \quad , \tag{4-11}$$

since ${}^{d\varepsilon}\!/_{d\tau} \approx 0$. Setting $\varepsilon = 0$ and substituting from (4-8) the fast dynamics is found to be

$$\frac{d}{d\tau}\begin{bmatrix} \varsigma_1 \\ \varsigma_2 \end{bmatrix} = \begin{bmatrix} 0 & I \\ -\overline{H}_{ff}\widetilde{K}_{ff} & 0 \end{bmatrix}\begin{bmatrix} \varsigma_1 \\ \varsigma_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \overline{B}_f^1 \end{bmatrix} u_F \,, \tag{4-12}$$

or

$$\frac{d\varsigma}{d\tau} = A_F\varsigma + B_F u_F \,, \tag{4-13}$$

with $\varsigma = \begin{bmatrix} \varsigma_1^T & \varsigma_2^T \end{bmatrix}^T$.

According to Tikhonov's theorem [14],[15] the original system (4-1) can be described to order ε using (4-9) and (4-13) with

$$\begin{aligned} q_r &= \overline{q}_r + O(\varepsilon) \\ q_f &= \varepsilon^2(\overline{\xi} + \varsigma_1) + O(\varepsilon) \end{aligned} \tag{4-14}$$

with $O(\varepsilon)$ denoting terms of order ε.

Now define the tracking output

$$y = \begin{bmatrix} \overline{q}_r \\ \dot{\overline{q}}_r \end{bmatrix} , \tag{4-15}$$

which corresponds to the slow part of the rigid-mode variables (e.g. of the link-tip motion). Assume that (A_F, B_F) is stabilizable, the fast system parameters have bounded uncertainties and perturbations (slow subsystem variables), and the slow system variables vary smoothly with time. The stabilizing assumption on (A_F, B_F) is satisfied in practical systems and is far milder that the requirement for stable zero dynamics. Moreover, the definition (4-15) corresponds to practical tracking objectives in contrast to the "reflected" outputs defined in [24],[37]. Under this assumptions a stabilizing control $u_F(t)$ can be designed using linear techniques (e.g. H_∞ design) so that

$$u_F = -\begin{bmatrix} K_{pF} & K_{dF} \end{bmatrix}\begin{bmatrix} \varsigma_1 \\ \varsigma_2 \end{bmatrix} = -\frac{K_{pF}}{\varepsilon^2} q_f - \frac{K_{dF}}{\varepsilon} \dot{q}_f + K_{pF}\bar{\xi} \tag{4-16}$$

stabilizes (4-13), with $\bar{\xi}$ given by (4-8).

4.2. Neural Net Control of the Rigid Dynamics

The slow dynamics given by (4-9) can be rewritten as

$$\overline{M}_{rr}\ddot{\bar{q}}_r + \overline{D}_{rr}\dot{\bar{q}}_r + \overline{F}_r + \overline{G}_r = \bar{u} \tag{4-17}$$

which is exactly the Lagrange form of an n-link rigid robot arm, satisfying the standard robot properties. For this part a neural network controller can be designed [17],[20]. Note that $\dot{\overline{M}}_{rr} - 2\overline{D}_{rr}$ is skew-symmetric.

Given a desired trajectory $q_d(t)$ for $\bar{q}_r$ the tracking error is

$$e = q_d - \bar{q}_r\,. \tag{4-18}$$

Define the filtered tracking error as

$$r = \dot{e} + \Lambda e\,, \tag{4-19}$$

where $\Lambda = \Lambda^T > 0$. Using (4-19), the arm dynamics can be rewritten in terms of the filtered tracking error as

$$\overline{M}_{rr}\dot{r} = -\overline{D}_{rr} r - \bar{u} + h(x) \tag{4-20}$$

where the nonlinear robot function is

$$h(x) = \overline{M}_{rr}(\bar{q})(\ddot{\bar{q}}_d + \Lambda\dot{e}) + \overline{D}_{rr}(\bar{q},\dot{\bar{q}})(\dot{\bar{q}}_d + \Lambda e) + \overline{F}_r(\dot{\bar{q}}) + \overline{G}_r(\bar{q}) \tag{4-21}$$

with $x = \begin{bmatrix} e^T & \dot{e}^T & \bar{q}_d^T & \dot{\bar{q}}_d^T & \ddot{\bar{q}}_d^T \end{bmatrix}^T$. It is assumed that *h*(*x*) is unknown.

A neural network can be used to estimate *h*(*x*) based on the universal approximation property of neural networks (2-12). This estimate is given by

$$\hat{h}(x) = \hat{W}^T \sigma(\hat{V}^T x), \tag{4-22}$$

let

$$Z \equiv \begin{bmatrix} V & 0 \\ 0 & W \end{bmatrix} \tag{4-23}$$

be the ideal weight matrix, which is unknown..

The functional approximation error of the neural network is

$$\tilde{h}(x) = h(x) - \hat{h}(x) \tag{4-24}$$

which can be written using a Taylor expansion, assuming smooth activation functions, as

$$\tilde{h}(x) = \tilde{W}^T (\hat{\sigma} - \hat{\sigma}' \hat{V}_h^T x) + \hat{W}^T \hat{\sigma}' \tilde{V}^T x + w, \tag{4-25}$$

where

$$\hat{\sigma} \equiv \sigma(V^T x), \qquad \hat{\sigma}' \equiv \left. \frac{\partial \sigma(z)}{\partial z} \right|_{z=\hat{z}}, \tag{4-26}$$

and the additional error term

$$w(t) = \tilde{W}^T \hat{\sigma}' V^T x + W^T O(\tilde{V}^T x)^2 + \varepsilon_{l_h}(x) \tag{4-27}$$

is bounded according to

$$w(t) \le C_0 + C_1 \|\tilde{Z}\| + C_2 \|x\| \|Z\| . \tag{4-28}$$

The jacobian $\hat{\sigma}'$ is an easily computed function of $\hat{V}^T x$.

It is assumed that the ideal weights of the neural network are bounded so that

$$\|Z\| \le Z_m \tag{4-29}$$

with Z_m a known bound, and the desired trajectory is bounded according to

$$\left\| \begin{matrix} q_d \\ \dot{q}_d \\ \ddot{q}_d \end{matrix} \right\| \le Q \tag{4-30}$$

with Q a known bound.

Definition

The solution to

$$\dot{x} = f(x,u,t), \qquad y = g(x,t)$$

is globally uniformly ultimately bounded (GUUB) if for all $x(t_0)$ there exists an $\varepsilon > 0$ and a number $T(\varepsilon, x_0)$ such that $\|x(t)\| < \varepsilon$ for all $t \ge t_0 + T$. □

Under all the assumptions stated above, a neural network controller is defined by the following theorem [40].

Theorem

Let the desired trajectory and the ideal unknown weights be bounded according to assumptions. Let the control input for (4-17) be defined by

$$\overline{u} = \hat{h} + K_v r - v \ , \qquad \text{for } K_v = K_v^T > 0 \tag{4-31}$$

with robustifying term

$$v(t) = -K_z(\|\hat{Z}\| + Z_m)r \tag{4-32}$$

and gain $K_z > C_2$.

Let the neural network weights be tuned by

$$\begin{aligned}\dot{\hat{W}} &= M(\hat{\sigma} - \hat{\sigma}'\hat{V}^T x)r^T - \kappa\|r\|M\hat{W} \\ \dot{\hat{V}} &= Nxr^T\hat{W}^T\hat{\sigma}' - \kappa\|r\|N\hat{V}\end{aligned} \tag{4-33}$$

with any constant matrices $M = M^T > 0,\ N = N^T > 0$, and a scalar design parameter $\kappa > 0$.

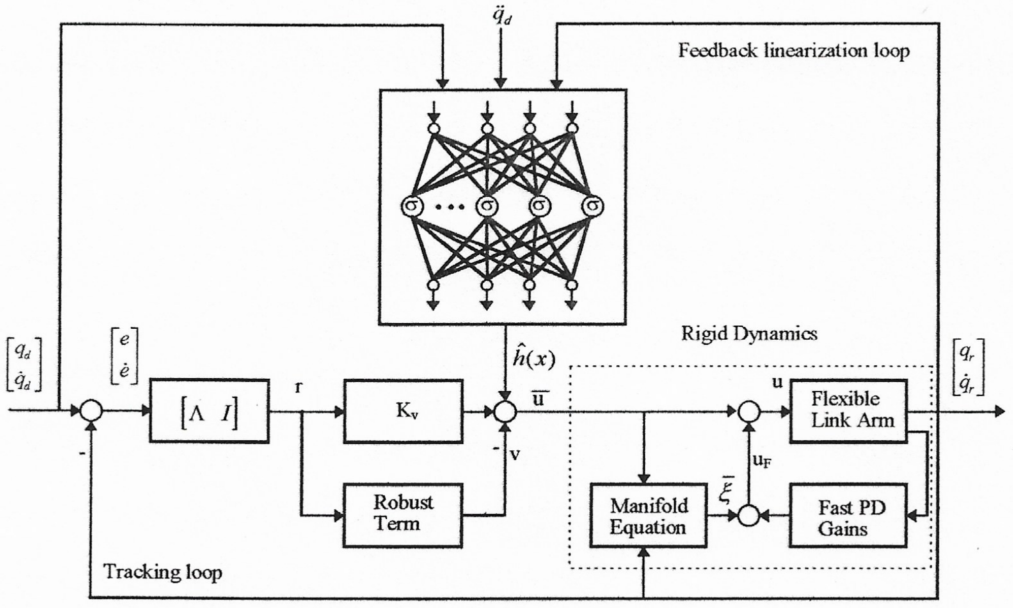
Then the filtered tracking error *r(t)* and the neural network weight errors $\tilde{V}$, $\tilde{W}$ are GUUB. Moreover, the tracking error may be kept as small as desired by increasing the gains K_v. □

The proof of this theorem uses Lyapunov theory and is given in [40], where explicit bounds on $\|r\|$ and $\|\hat{Z}\|$ are given. Notice that the training rules in

(4-33) include the standard backpropagation terms plus an e-modification [27] and a second-order correction term. Furthermore, the NN weights can be easily initialized at zero since the PD control stabilizes the system while the NN is learning. The NN controller is designed to control the robot arm while it is learning to improve the performance, hence no off line training is required.

4.3. Overall Control Structure

The overall structure of the controller defined in sections 4.1 and 4.2 is shown in figure 6.



$\ddot{q}_d$
Feedback linearization loop
$\begin{bmatrix} q_d \\ \dot{q}_d \end{bmatrix}$
$\begin{bmatrix} e \\ \dot{e} \end{bmatrix}$
$[\Lambda \ I]$
r
K$_v$
$\hat{h}(x)$
$\overline{u}$
v
Robust Term
Rigid Dynamics
u
Flexible Link Arm
u$_F$
$\overline{\xi}$
Manifold Equation
Fast PD Gains
$\begin{bmatrix} q_r \\ \dot{q}_r \end{bmatrix}$
Tracking loop

Fig. 6. Overall control structure of the neural network controller for a flexible link robot.

4.4. Simulation

The simulation of the neural network controller was performed for a single flexible link. The model of the flexible link included three flexible modes even though the controller only compensated the first two modes (this was to corroborate that the controller work well even compensating only a finite number of modes). The model was obtained as described in [10], using the parameters of the flexible link test-bed at the Automation and Robotics Research Institute (ARRI). The modal frequencies for the first three modes for this flexible link are 1.6Hz, 10.0Hz, and 28.1Hz.

The controller used the following parameters:

$$
\begin{aligned}
K_v &= 36 \\
\Lambda &= \frac{200}{36} \\
K_z &= 0.2 \\
Z_m &= 50 \\
\frac{K_{pf}}{\varepsilon^2} &= \begin{bmatrix} -8 & 10 \end{bmatrix} \\
\frac{K_{df}}{\varepsilon} &= \begin{bmatrix} 0 & 0 \end{bmatrix}
\end{aligned}
$$

The neural network in the controller included ten neurons in the hidden layer and used the following parameters

$$
\begin{aligned}
F &= 20 \\
G &= 20 \\
\kappa &= 0.000001
\end{aligned}
$$

The activation functions for the neurons in the hidden layer were selected as the sigmoid functions

$$\sigma_k(z) = \frac{1}{1+e^{-k\alpha z}} \qquad \text{for } k = 1,\ 2,\ldots,\ 10;\quad \alpha = 1$$

It was observed in practice that with the sigmoid functions defined this way, the neural network learned faster and was able to reduce more the tracking error.

The results of the simulation are plotted in figure 7. Notice that after some time the neural net learns the model of the link reducing the tracking error to almost zero. Using a bigger value for the learning rates F and G improved the tracking performance (faster learning and less tracking error), but produced a worse transient response (more oscillatory) and more excitation of the flexible modes (increased the magnitude of q_{f1}).

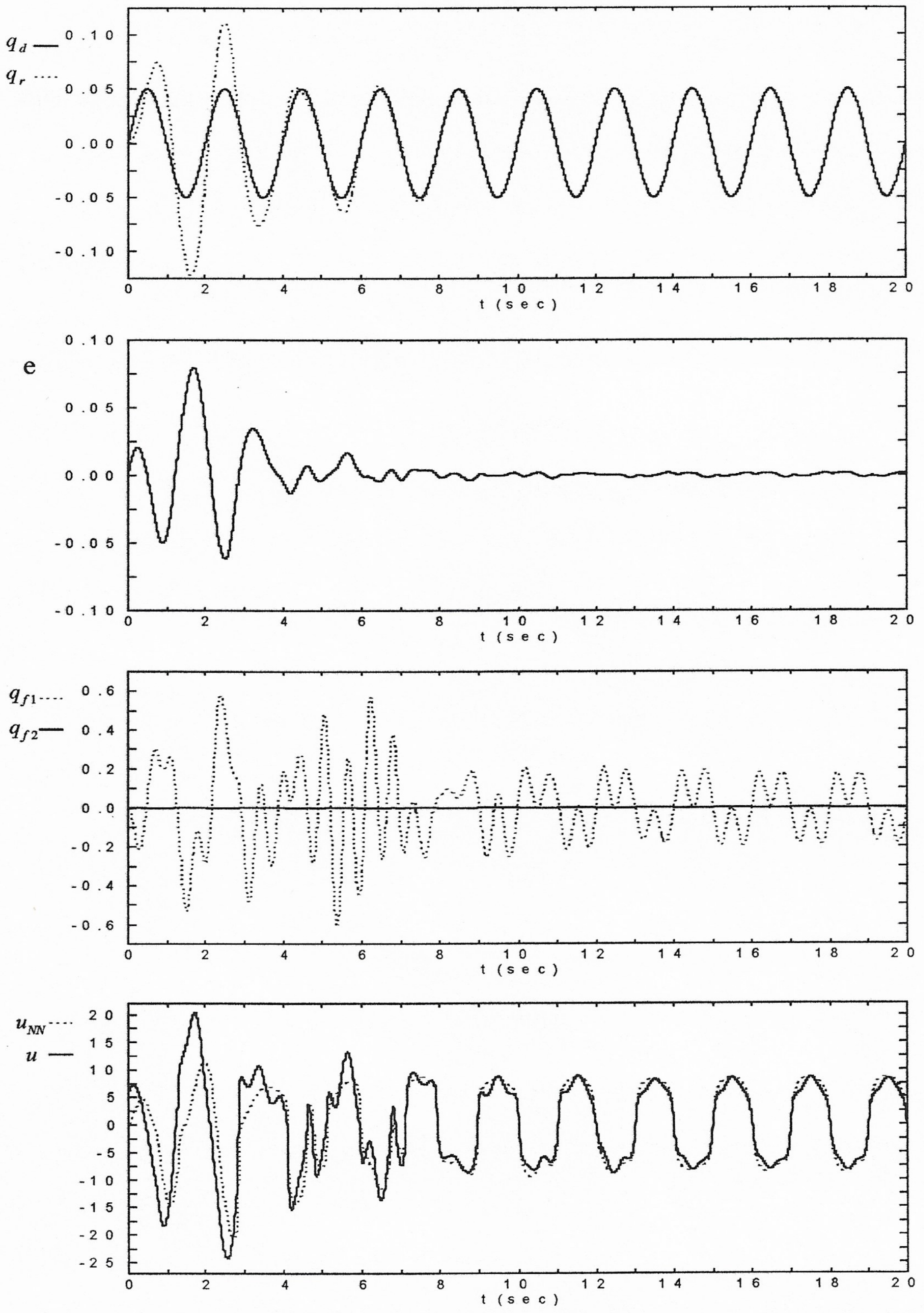


Fig. 7. Simulation of the neural network controller with a single flexible link.

CHAPTER 5

IMPLEMENTATION OF THE CONTROLLER IN THE FLEXIBLE-LINK TEST-BED

The neural network controller discussed in chapter 4 was implemented on a single flexible link test-bed at the ARRI and some of the results obtained are presented here.

5.1. Description of the Implementation

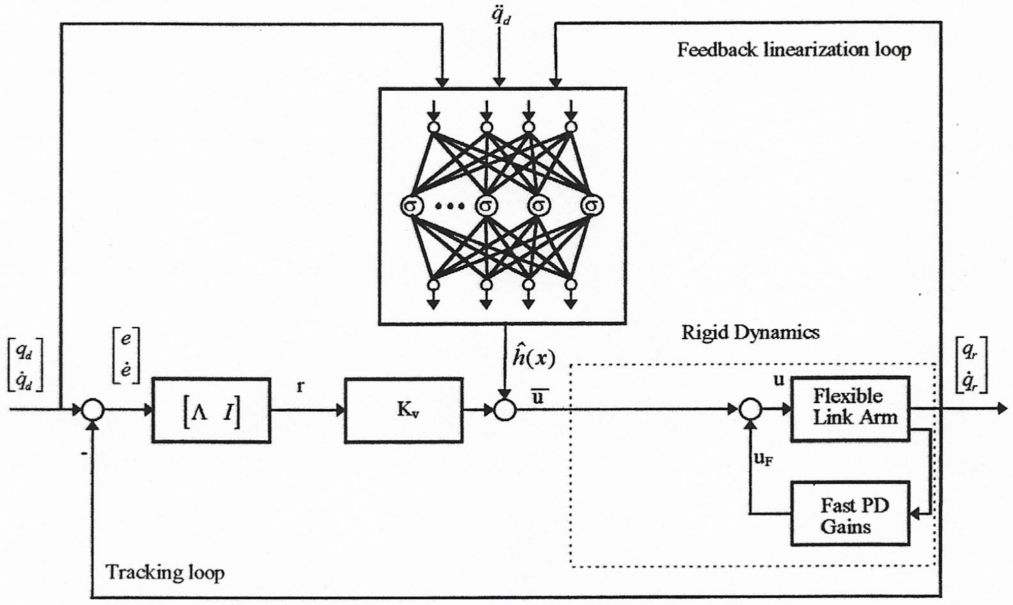
The actual structure of the controller implemented in the test-bed at the ARRI is shown in figure 8. A list of the main characteristics of the practical implementation is given bellow:

- Only the first two flexible modes were considered.
- The robust term v was not included. K_v was selected big enough to avoid the necessity of v. The manifold term was not included since the actual model of the flexible link is unknown. As shown by (4-8), the implementation of $\bar{\xi}$ would require the exact knowledge of the matrices of the model.
- Even though the dynamics for the flexible link with one degree of freedom is linear, an extra nonlinear friction term was added to check the capability of the controller to compensate for the nonlinearities in the model.
- The neural network is composed of ten neurons in the hidden layer, with five inputs ($x = \begin{bmatrix} e & \dot{e} & \bar{q}_d & \dot{\bar{q}}_d & \ddot{\bar{q}}_d \end{bmatrix}^T$) and one output $\hat{h}(x)$.

- The controller defined by (4-5), (4-16), (4-31), and

- (4-33) was discretized with sampling period of 5ms. In the discretization process the differential equations in

- (4-33) were solved on line using trapezoidal integration.



$\ddot{q}_d$
Feedback linearization loop
$\begin{bmatrix} q_d \\ \dot{q}_d \end{bmatrix}$
$\begin{bmatrix} e \\ \dot{e} \end{bmatrix}$
$[\Lambda \ I]$
r
K_v
$\hat{h}(x)$
$\overline{u}$
Rigid Dynamics
u
Flexible Link Arm
u_F
Fast PD Gains
$\begin{bmatrix} q_r \\ \dot{q}_r \end{bmatrix}$
Tracking loop

Fig. 8. Actual neural network controller implemented at ARRI's flexible link test-bed.

Figure 9 shows a picture of the flexible link controller implementation at ARRI. A block diagram describing the practical implementation of the controller is shown in figure 10. The hardware includes the interface cards and external components necessary for the measurement of the angular position of the link q_r and the flexible modes q_{f1} and

q_{f2} (optical encoder, strain gauges, signal conditioners, and analog to digital converters). Estimated values of $\dot{q}_r$, $\dot{q}_{f1}$, and $\dot{q}_{f2}$ are calculated based on consecutive samples of q_r, q_{f1}, and q_{f2} respectively. Besides there is a digital to analog converter connected to the servo amplifier that drives the servo-motor for the link.

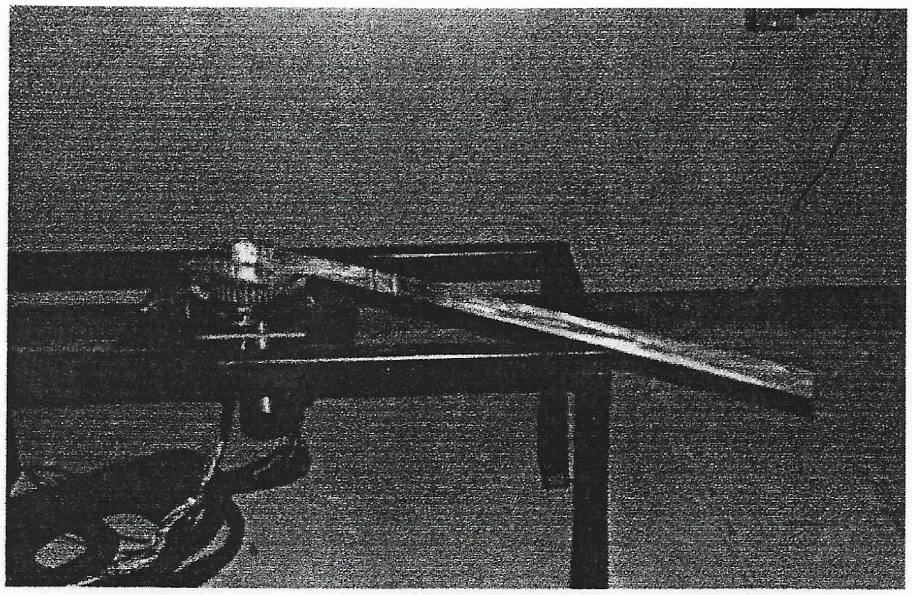


Fig. 9. Flexible link controller implementation at ARRI.

The software was implemented in LabView and C. The routines that perform the control action in real time are implemented in C. The execution of these external routines is fired periodically by the computer timer routines. The control routines sample the external signals and use the parameters defined in the parameters buffer to calculate the control signal u. Some of the signals are stored in the signals buffer allowing the LavView

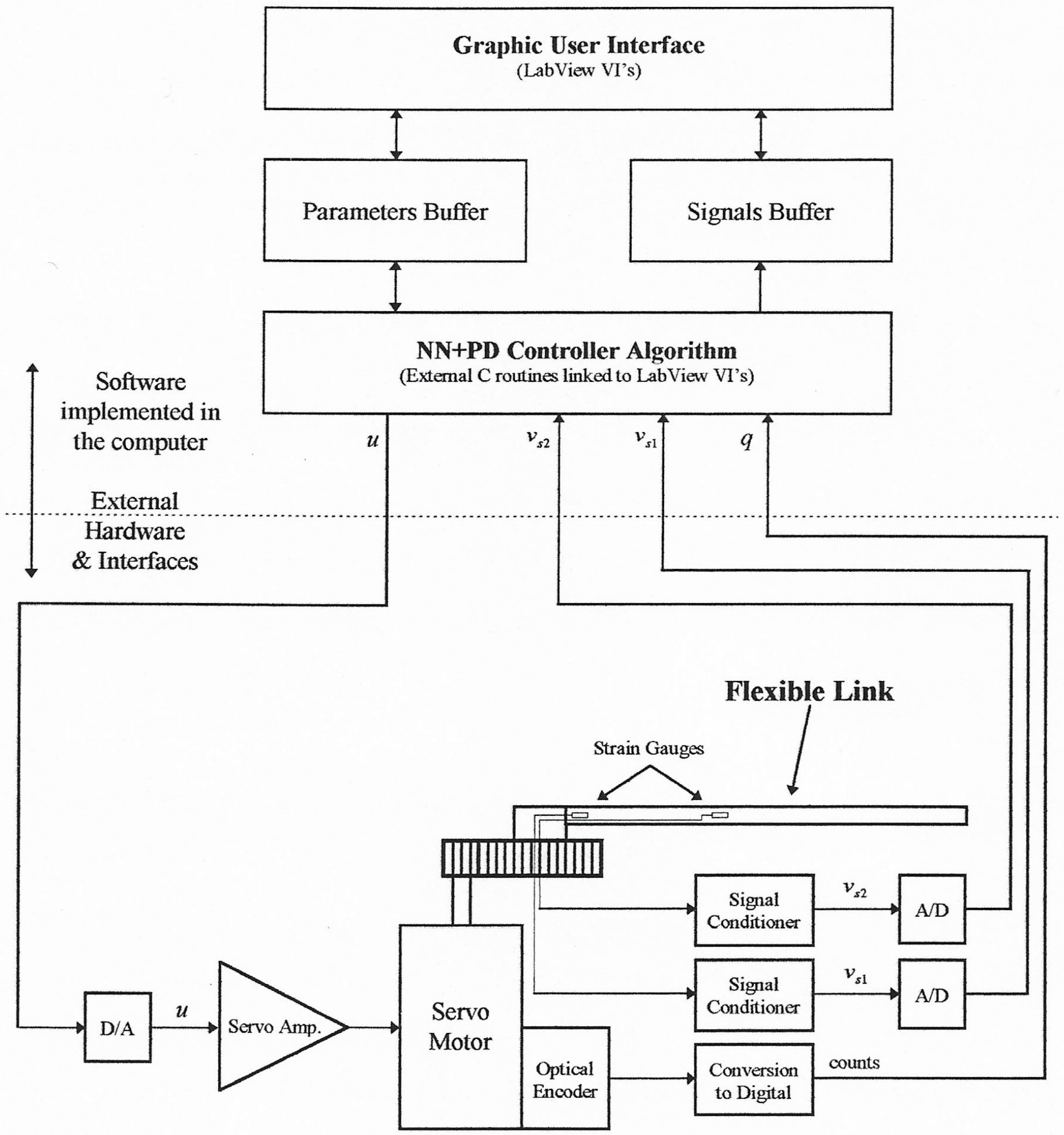


Fig. 10. Block diagram of neural network controller implementation at ARRI's flexible link test-bed.

VI's to monitor them. A listing of the C code of the fuctions more relevant in the inplementation of the NN controller is given in appendix A.

The LabView VI's work as a graphic user interface that allows to start the controller, change the mode of operation, define the reference signals, change the parameters of the controller, and monitor the signals through charts and graphics. These VI's are linked to the external C routines which run in the background in real time. The communication between the external C routines and the LabView VI's is accomplished through some VI's that read from and write to the buffers using CIN's (Code Interface Nodes).

5.2. Experimental Results in the Flexible Link Test-bed

Standard controllers PD and PID were implemented and tested in the flexible link test-bed to compare their performance with the PD+NN (PD and Neural Net) controller. This comparison allows us to show the advantages of the proposed controller over the standard controllers.

5.2.1. PD control

A PD controller was implemented using the control law

$$u = \overline{u} + u_F$$

with

$$\overline{u} = K_v r = K_v(\dot{e} + \Lambda e)$$

$$u_F = -\begin{bmatrix} K_{pF} & K_{dF} \end{bmatrix} \begin{bmatrix} q_f \\ \dot{q}_f \end{bmatrix}$$

using the following parameters

$$K_v = 36$$

$$\Lambda = \frac{200}{36}$$

$$K_{pf} = \begin{bmatrix} -8 & 10 \end{bmatrix}$$

$$K_{df} = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

and the reference signal

$$q_d = 0.05Sin(2\pi ft)$$

with frequency f=0.5Hz.

The performance of the tracking PD control without the neural network is illustrated in figure 11(a) without the extra friction term, and figure 11(b) with the extra friction term. Notice that the tracking error is very big, its magnitude is comparable to that of the reference signal. Even though the magnitude of the error decreases incrementing the controller gains, the tracking error is not eliminated. These characteristics are preserved in the presence of the extra friction term.

5.2.2. PID Control

A PID controller was implemented using the control law

$$u = \bar{u} + u_F$$

with

$$\bar{u} = K_v r + K_i \int e\,dt = K_v(\dot{e} + \Lambda e) + K_i \int e\,dt$$

$$u_F = -\begin{bmatrix} K_{pF} & K_{dF} \end{bmatrix} \begin{bmatrix} q_f \\ \dot{q}_f \end{bmatrix}$$

using the following parameters

$$\begin{aligned} K_v &= 36 \\ \Lambda &= \frac{200}{36} \\ K_i &= 100 \\ K_{pf} &= \begin{bmatrix} -8 & 10 \end{bmatrix} \\ K_{df} &= \begin{bmatrix} 0 & 0 \end{bmatrix} \end{aligned}$$

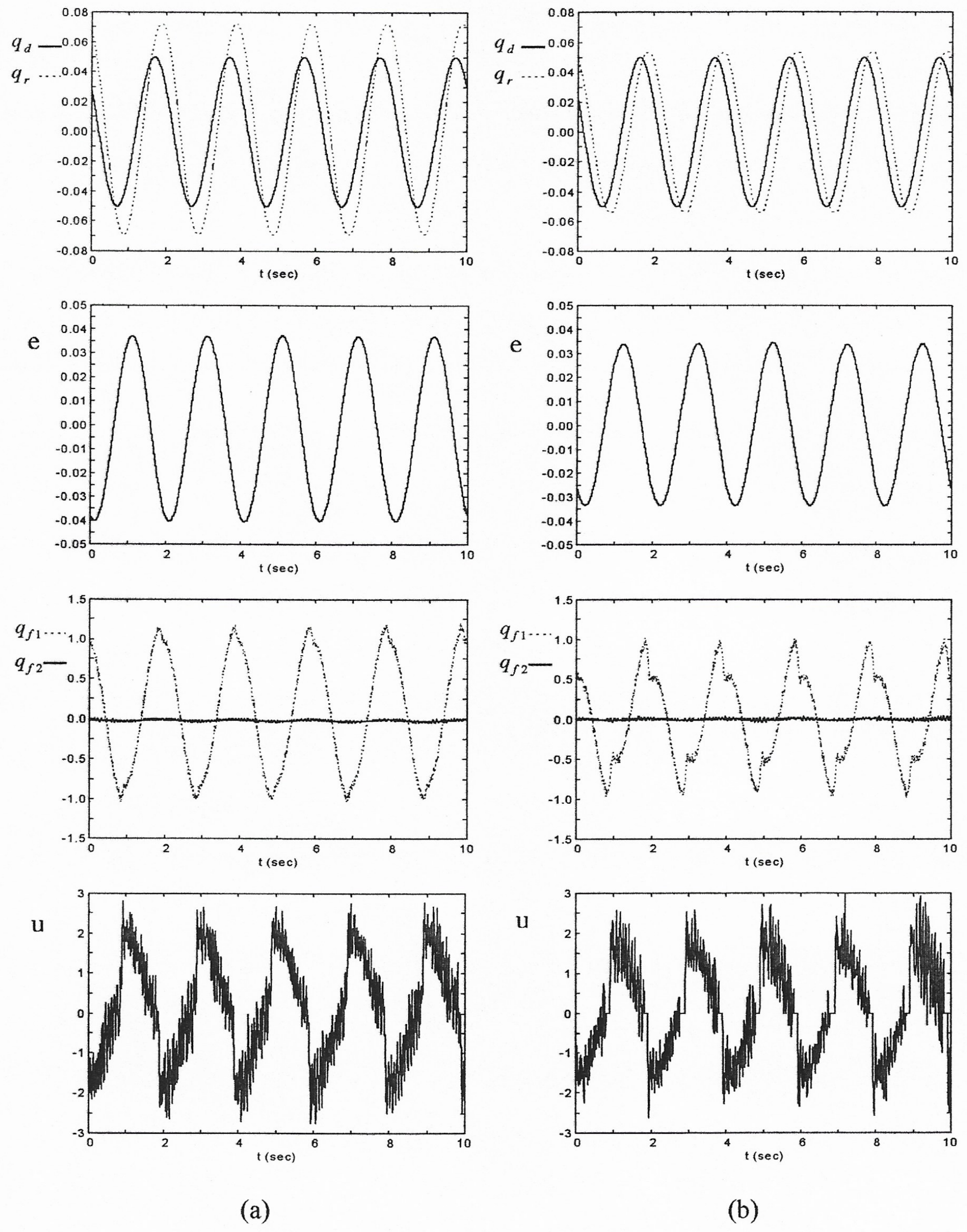


Fig. 11. Performance of the PD control. (a) Without additional friction. (b) With additional friction.

and the reference signal

$$q_d = 0.05Sin(2\pi ft)$$

with frequency f=0.5Hz.

The performance of the tracking PID control is illustrated in figure 12(a) without the extra friction term, and figure 12(b) with the extra friction term. The integral part of the PID controller is supposed to eliminate the steady state error, but that only works for constant desired trajectories. In this case, with a varying desired trajectory, the tracking is even worse when the integral part is introduced (notice that the tracking error is bigger than with the PD control). As in the case of the PD controller, the PID controller is not able to compensate for the extra friction term.

5.2.3. NN+PD Control

The neural net tracking controller was implemented as described in section 5.1 using the same parameters of the simulation in section 4.4, except that in this case

$$F = 2$$
$$G = 20$$
$$\kappa = 0.000001$$

This value of F in the practical implementation was enough. A bigger value produced a response very oscillatory.

The reference signal was

$$q_d = 0.05Sin(2\pi ft)$$

with frequency f=0.5Hz.

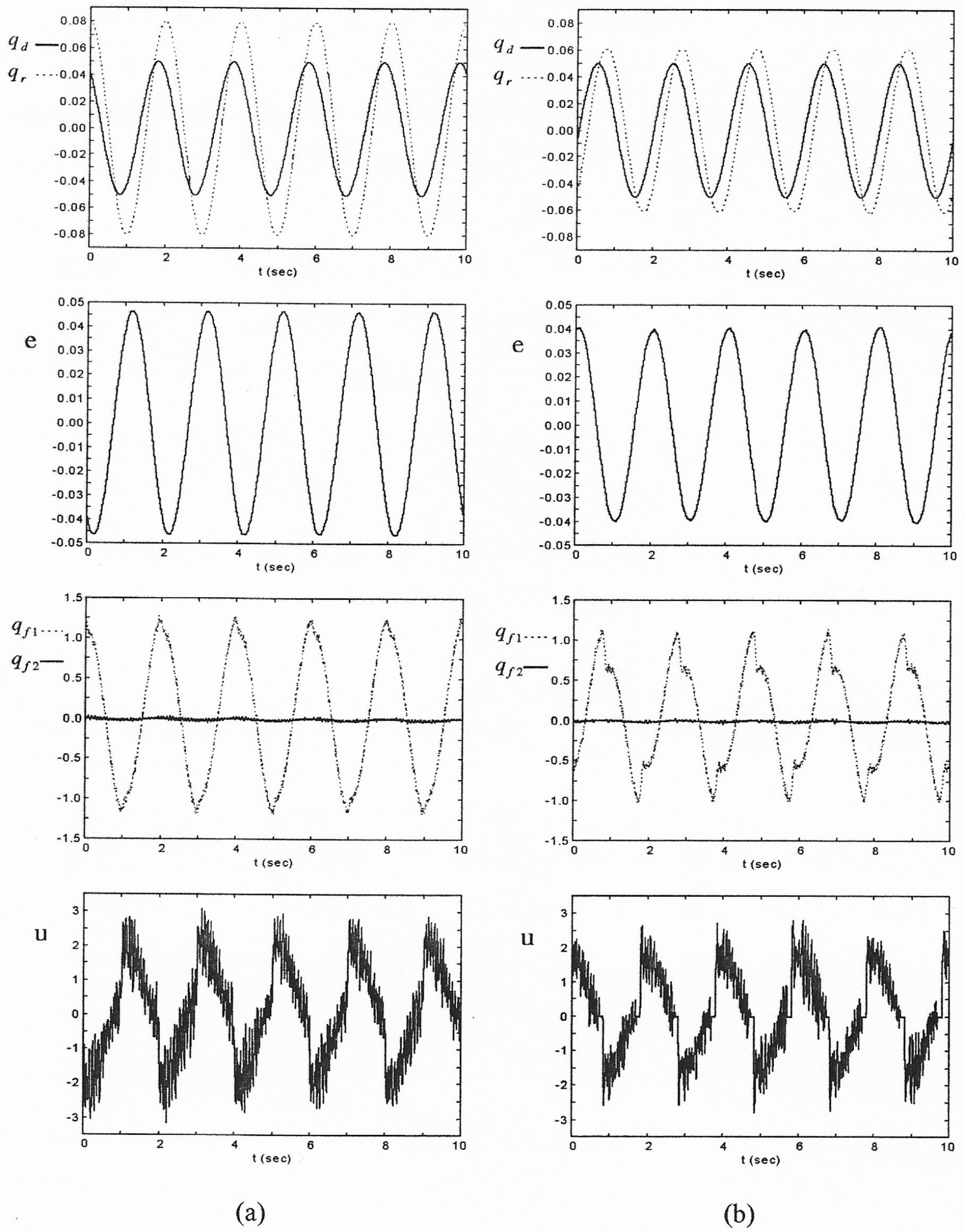


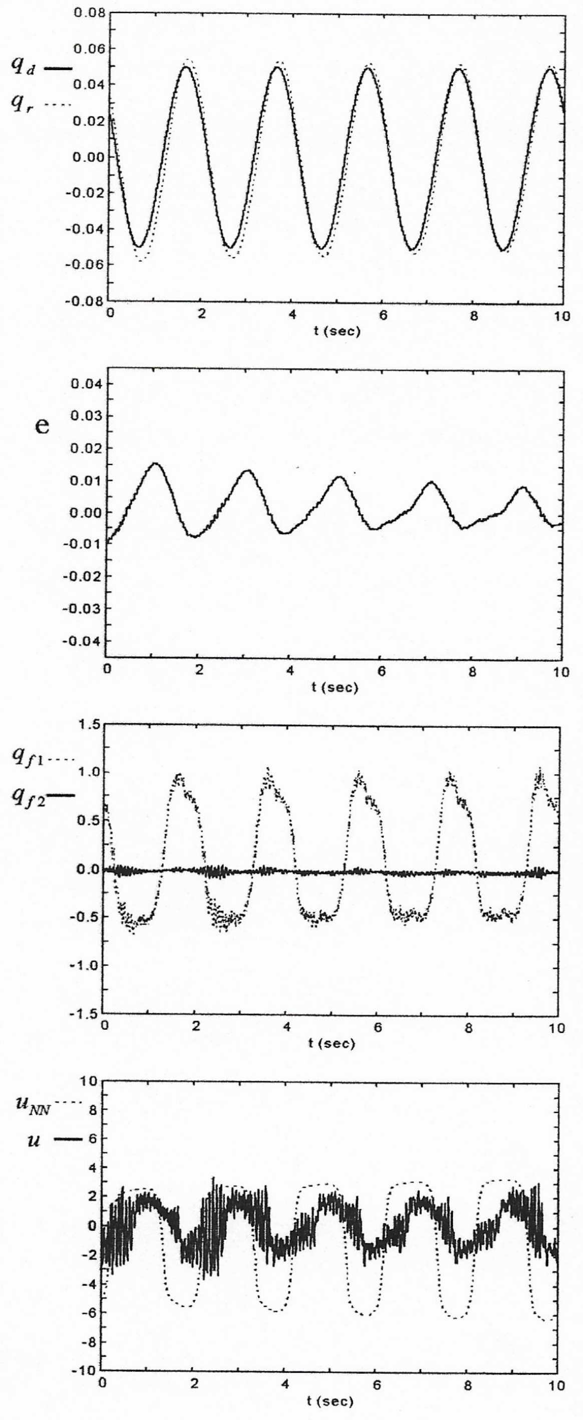
Fig. 12. Performance of the PID control. (a) Without additional friction. (b) With additional friction.

The performance of the neural network tracking control is illustrated in figure 13 before the learning is complete and in figure 14 after the learning is complete. The training of the neural network take less than one minute, after which the tracking error is reduced to almost zero. The learning is really active all the time (on-line training), but we refer to learning complete to the instant when the neural network have learned the model of the link under the actual conditions. Notice in figure 14(a), without the extra friction term, and figure 14(b), with the extra friction term, that the same controller learns the model of the link readapting to changes in it (changes in the model like changes in friction characteristics). Without changing the parameters of the controller, the neural network controller is able to take the tracking error to almost zero in both cases.

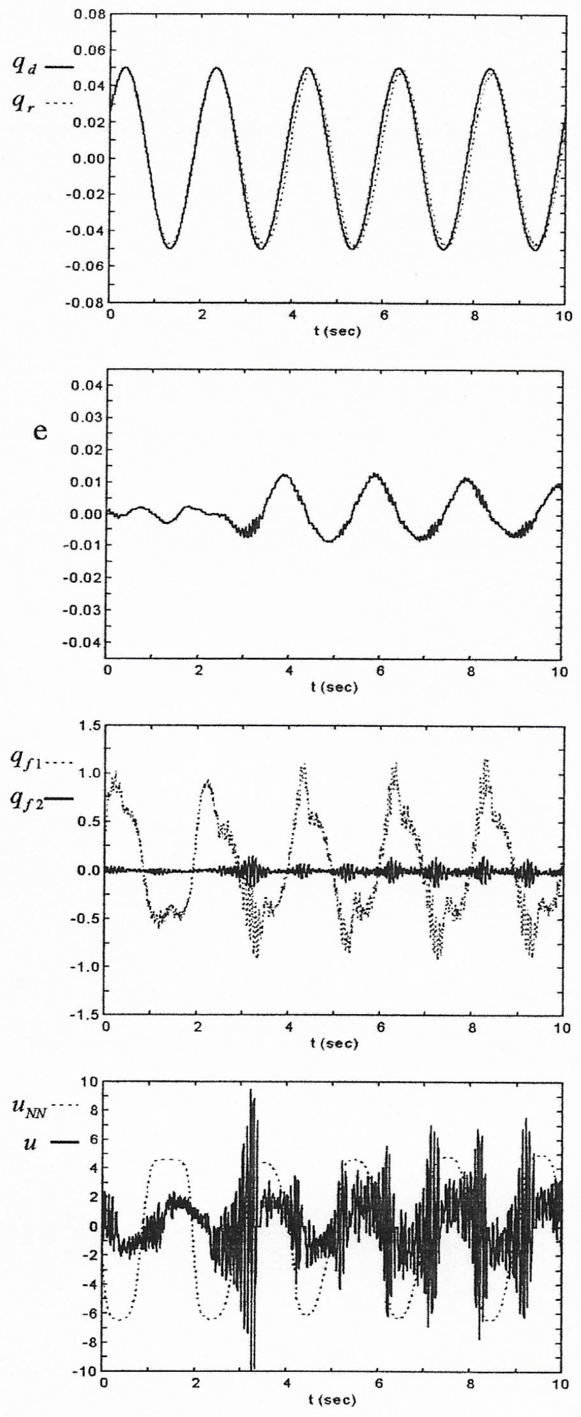
In practice it was noticed that a change in the reference signal increased the tracking error momentarily requiring a readaptation of the neural network. However after some time, when the neural network learned the new conditions, it was able to get rid of the tracking error.

5.3. Comparison between different approaches

Comparing the tracking performance of the different controllers shown in figure 11 for the PD controller, figure 12 for the PID controller, and figure 13 and figure 14 for the neural net controller, it is clear the superiority of the last one. Even the PID cannot be a better tracking controller than the neural network controller under a varying desired trajectory. In this situation the PD controller is better than the PID, but not as good as the neural network controller.



(a)



(b)

Fig. 13. Performance of the PD+NN control before learning. (a) Without additional friction. (b) With additional friction.

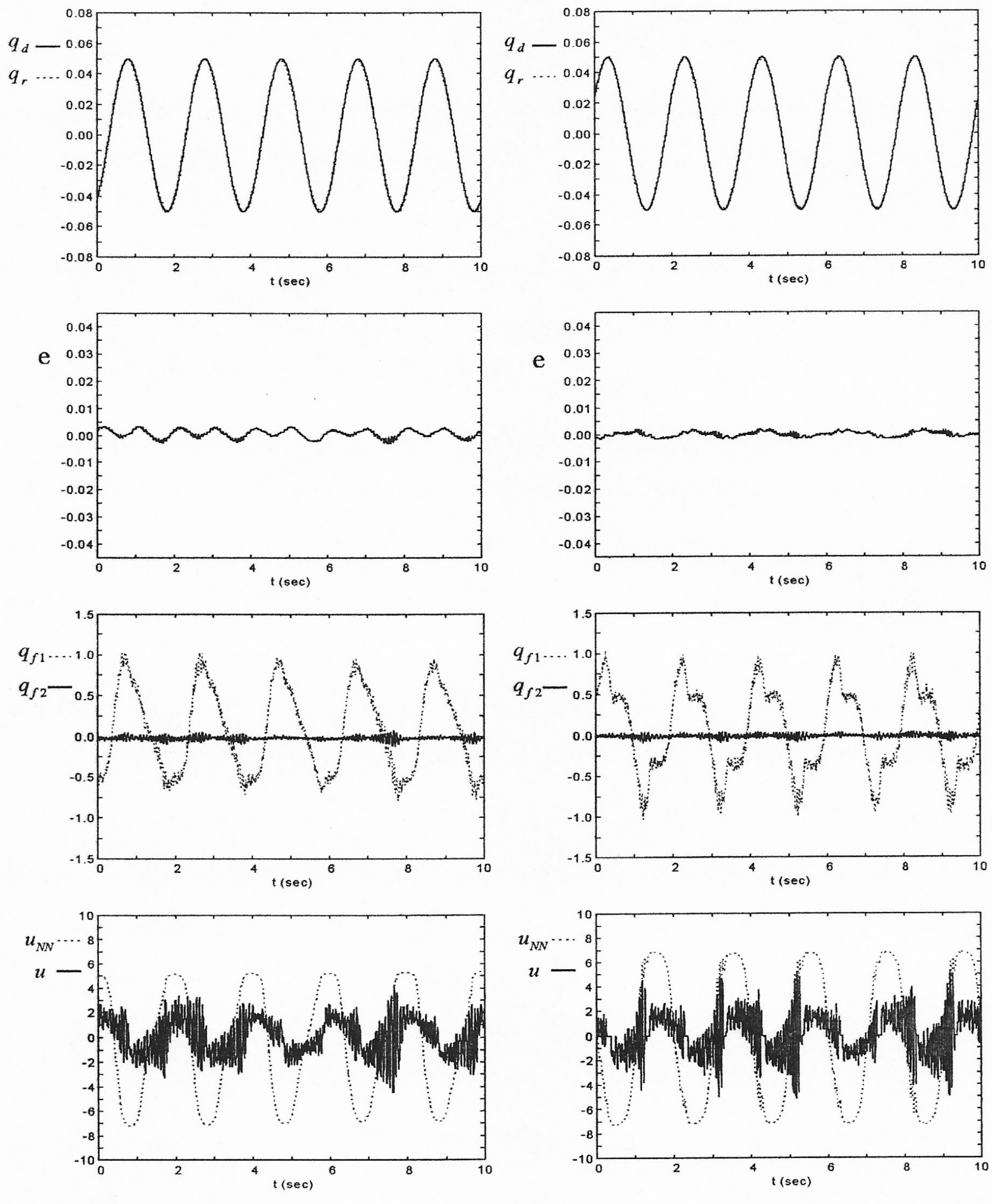


Fig. 14. Performance of the PD+NN control after learning. (a) Without additional friction. (b) With additional friction.

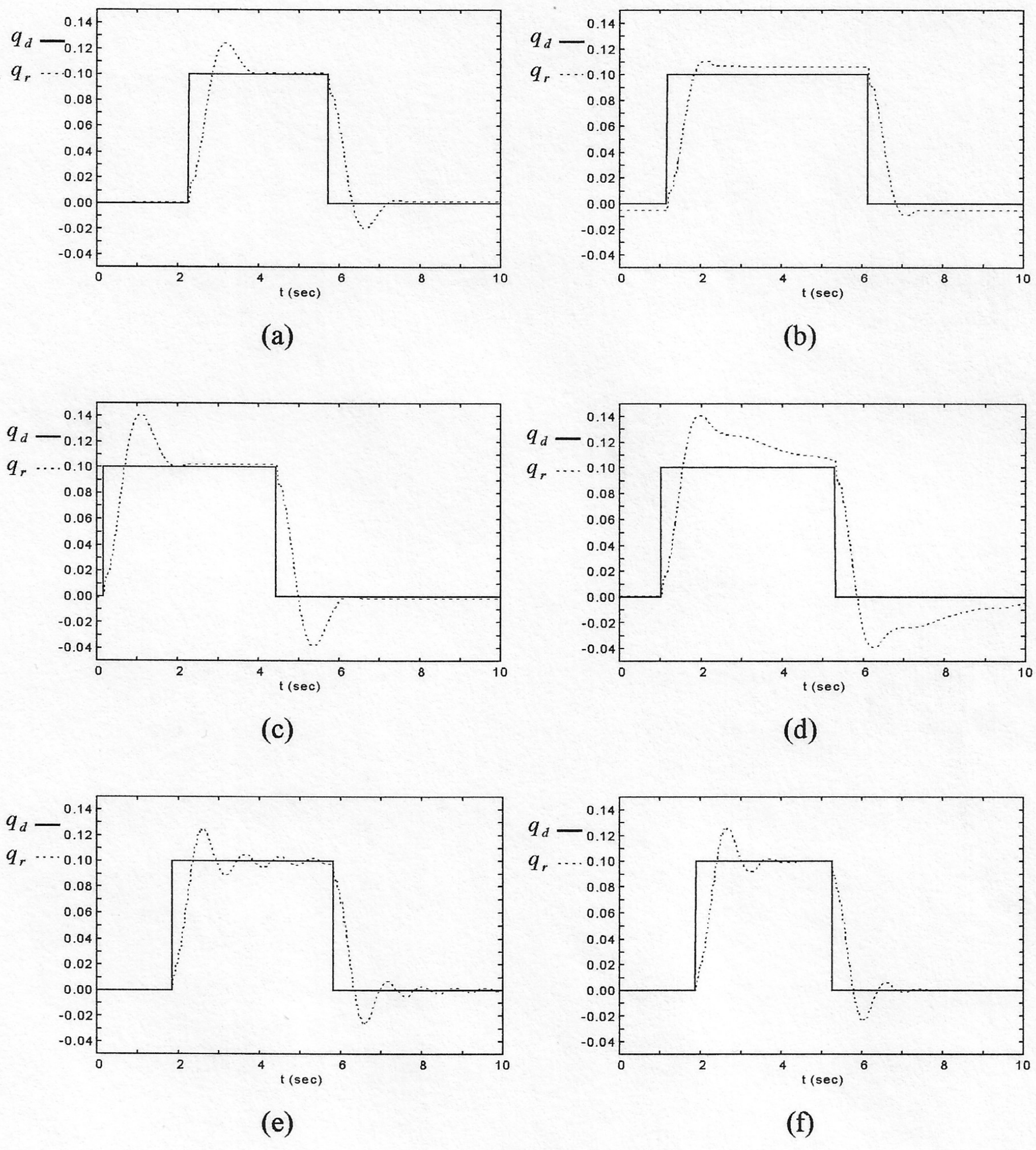


Fig. 15. Step response of the controllers. (a) PD control. (b) PD control with extra friction. (c) PID control. (d) PID control with extra friction. (e) PD+NN control. (f) PD+NN control with extra friction.

CHAPTER 6

CONCLUSIONS

The practical implementation of a multiloop nonlinear neural network tracking controller for a single flexible link has been tested and its performance compared to the one of the standard PD and PID controllers. An extra friction term was added in the implementation to show the ability of the neural network controller to learn and compensate for the nonlinearities.

The controller includes an outer PD tracking loop, a singular perturbation inner loop for stabilization of the fast dynamics, and a neural network inner loop used to feedback linearize the slow dynamics. This NN controller requires no off-line learning phase, the NN weights are easily initialized, and guarantees boundedness of the tracking error and control signal.

The practical results corroborate the simulations showing that standard PD or PID controllers are not able to track a varying desired trajectory, while the neural network controller takes the tracking error to almost zero readapting to any changes in the model of the link (extra friction terms).

APPENDIX A

C CODE OF THE FUNCTIONS MORE RELEVANT IN THE INPLEMENTATION OF THE NN CONTROLLER

```
/*
--------------------------------------------------------------------------------
NNPIDFlexLink.h

Andy Lowe, 2/22/95
Copyright 1993 Andy Lowe, University of Texas at Arlington
--------------------------------------------------------------------------------

Modified for Neural Net & PID Control by Luis Gutierrez, 4/9/96

--------------------------------------------------------------------------------
*/

#ifndef _CTRL
#define _CTRL

#include "FnSched.h"
#include "MIO16.h"
#include "nuControl.h"

#define N_FUNCS 5
struct sFuncs {                 /*      function ID numbers */
        Int16   Sample;         /*      sample y and (optionally) tach */
        Int16   Ctrl;           /*      control algorithm */
        Int16   Monitor;        /*      signal monitor */
        Int16   Cmd;            /*      command processor */
        Int16   Watchdog;       /*      watchdog timer */
};
typedef struct sFuncs sFuncs;

#define N_BUFS 6
struct sBufs {                  /*      buffer ID numbers */
        Int16   CmdQ;           /*      command queue */
        Int16   InputQ;         /*      signal input queue */
        Int16   MonitorQ;       /*      signal monitor output queue */
        Int16   Wave;           /*      periodic waveform buffer */
        Int16   Config;         /*      configuration buffer */
        Int16   Parms;          /*      parameter buffer */
};
typedef struct sBufs sBufs;

#define CMD_HDR                                                                 \
        Int32   Size;           /*      number of bytes which follow */         \
        Int16   Cmd;            /*      command opcode */
struct sCmdQ {                  /*      command queue element header */
        CMD_HDR
};
typedef struct sCmdQ sCmdQ;
#define CMDQSIZE 2048           /*      size of command queue in bytes */

typedef Flt sInputQ;            /*      input signal queue element */

struct sMonitorQ {              /*      signal monitor output queue elements */
        Flt     r;              /*      reference input */
        Flt     y;              /*      position measurement */
        Flt     u;              /*      control signal */
        Flt     v;              /*      derivative of measurement or tach */
        Flt qf1;                /*      deflection estimate 1 (in) */
        Flt qf2;                /*      deflection estimate 2 (in) */
        Flt dqf1;               /*      deflection rate 1 (in/s) */
        Flt dqf2;               /*      deflection rate 2 (in/s) */
        Flt vs1;                /*      strain gauge 1 voltage */
        Flt vs2;                /*      strain gauge 2 voltage */
};
typedef struct sMonitorQ sMonitorQ;

typedef Flt sWave;              /*      waveform table element */

struct sConfig {                /*      hardware configuration  parameters */
        sMIOConfig      MIOConfig;      /* MIO-16(X) configuration structure */
        Int16           strain1Chan;    /* A/D channel number for strain 1 measurement */
        Int16           strain1Gain;    /* instrumentation amp gain for strain 1 */
        Flt     Vfrcw;          /* viscous  friction magnitude in CCW direction (V*s) or
(V/RPM) */
        Int16   EnIntegral;     /* Enable integral term */

        /******************* Parameters for Neural Net   ***********************/

        Flt             Kappa;          /* neural net e-mod rate */
        Flt             F;              /* neural net learning rate for W  */
        Flt             G;              /* neural net learning rate for V  */
        Int16           addNeural;      /* addNeural button */

        /**********************************************************************/

};
typedef struct sCtrlParms sCtrlParms;

struct sSigLimits {             /* signal limits */
        LVBool          rLimit; /* true-->limit reference signal */
        Flt             rMax;   /* maximum reference signal */
        Flt             rMin;   /* minimum reference signal */
        LVBool          uLimit; /* true-->limit control signal */
        Flt             uMax;   /* maximum control signal */
        Flt             uMin;   /* minimum control signal */
};
typedef struct sSigLimits sSigLimits;

struct sRateLimit {             /* limit rate of change of reference signal */
        LVBool  rateLimit;      /* true-->limit reference signal rate */
        Flt     rateMax;        /* maximum rate magnitude (units / sec) */
};
typedef struct sRateLimit sRateLimit;

struct sFolError {              /* following error */
        LVBool  eLimit;         /* true-->limit following error */
        Flt     eMax;           /* maximum following error magnitude */
};
typedef struct sFolError sFolError;

struct sWaveParms {             /* periodic waveform parameters */
        Flt     Freq;           /* frequency (Hz) */
        Flt     Amp;            /* amplitude */
        Flt     Offset;         /* offset */
};
typedef struct sWaveParms sWaveParms;

struct sSigMonitor {            /* signal monitor */
        Int32   i;              /* iteration */
        sMonitorQ Sig;          /* signals */
};
typedef struct sSigMonitor sSigMonitor;

struct sParms {                 /* parameter buffer structure */
        sMode           Mode;
        sCtrlStatus     CtrlStatus;
        sCmdResult      CmdResult;
        sFuncPrd        FuncPrd;
        sCtrlParms      CtrlParms;
        sSigLimits      SigLimits;
        sRateLimit      RateLimit;
        sFolError       FolError;
        sWaveParms      WaveParms;
        sSigMonitor     SigMonitor;
};
typedef struct sParms sParms;
#define MODE            0
#define CTRLSTATUS      sizeof(sMode)
#define CMDRESULT       (CTRLSTATUS + sizeof(sCtrlStatus))
#define FUNCPRD         (CMDRESULT + sizeof(sCmdResult))
#define CTRLPARMS       (FUNCPRD + sizeof(sFuncPrd))
#define SIGLIMITS       (CTRLPARMS + sizeof(sCtrlParms))
#define RATELIMIT       (SIGLIMITS + sizeof(sSigLimits))
#define FOLERROR        (RATELIMIT + sizeof(sRateLimit))
```

```
#define WAVEPARMS       (FOLERROR + sizeof(sFolError))
#define SIGMONITOR      (WAVEPARMS + sizeof(sWaveParms))
        /*      command opcodes */
#define NO_OP           0x0000  /* no operation */
#define WR_MODE         0x0101  /* write mode */
#define WR_FUNCPRD      0x0102  /* write function periods */
#define WR_CTRLPARMS    0x0103  /* write control parameters */
#define WR_SIGLIMITS    0x0104  /* write signal limits */
#define WR_RATELIMIT    0x0105  /* write rate limit */
#define WR_FOLERROR     0x0106  /* write following error limit */
#define WR_WAVEPARMS    0x0107  /* write waveform parameters */

struct sCmds {
        CMD_HDR
        union {
                sMode           Mode;
                sFuncPrd        FuncPrd;
                sCtrlParms      CtrlParms;
                sSigLimits      SigLimits;
                sRateLimit      RateLimit;
                sFolError       FolError;
                sWaveParms      WaveParms;
        }       Data;
};
typedef struct sCmds sCmds;

/********************* Neural Net constant definitions ***********************/

#define N1 5            /* # of NN Inputs */
#define N2 10           /* # of Hidden Layer Outputs */
#define N3 1            /* # of Output */

#define NN_INPUT_e   0
#define NN_INPUT_ed  1
#define NN_INPUT_r   2
#define NN_INPUT_vrk 3
#define NN_INPUT_ark 4

#define satFlt(x) x             /*   MODIFY THIS TO SATURATE WEIGHTS IN NN   */

#if !__option(a4_globals)
#define DSNewPtr malloc
#define DSDisposePtr free
#endif

/*****************************************************************************/

/*      prototypes */
/*  NNFlexLink.new.c  */

LVBool InstallCtrl(Int32 inputQSize, Int32 monitorQSize, Int32 waveSize);
Void RemoveCtrl(Void);
LVBool ActivateCtrl(Void);
Void DeactivateCtrl(Void);
Void GetBufIDs(sBufs *bufs);
Void GetCtrlState(LVBool *isInstalled, LVBool *isActive);
Void EStop(Void);
Void Sample(TMFuncList *fl);
Void Ctrl(TMFuncList *fl);
Void Monitor(TMFuncList *fl);
Void Cmd(TMFuncList *fl);
Void Watchdog(TMFuncList *fl);

/****************************** Neural Net prototypes ************************/
/*  NeuralNet.c  */

Int16 AllocateNN(Void);
Void DisposeNN(Void);
Flt NeuralNetCtrl(Flt x[] , Flt fltre , Flt samplePrd, const sCtrlParms *ctrlParms);

/*****************************************************************************/
#endif
```

```
/*
---------------------------------------------------------------------------------
NNPIDFlexLink.c

Flexible link controller.

Andy Lowe, 2/22/95
Copyright 1993 Andy Lowe, University of Texas at Arlington
---------------------------------------------------------------------------------

Modified for Neural Net & PID Control by Luis Gutierrez, 4/9/96

---------------------------------------------------------------------------------
*/

/* Include Files */
#include "NNPIDFlexLink.h"

pascal void Debugger(void)
       = 0xA9FF;
pascal void DebugStr(const char *)
       = 0xABFF;

/*     initialize parameters */

static sFuncs Funcs = {
       -1,             /*      Sample */
       -1,             /*      Ctrl */
       -1,             /*      Monitor */
       -1,             /*      Cmd */
       -1,             /*      Watchdog */
};

static sBufs Bufs = {
       -1,             /*      CmdQ */
       -1,             /*      InputQ */
       -1,             /*      MonitorQ */
       -1,             /*      Wave */
       -1,             /*      Config */
       -1,             /*      Parms */
};

static sConfig Config = {
       {                       /* MIOConfig */
               0xd,            /* slot */
               0,              /* mio16x */
               mio16x18,       /* mio16xClock */
               0,              /* uniADC */
               20.0,           /* rangeADC */
               0,              /* uniDAC */
               10.0,           /* refDAC */
       },
       0,                      /* strain1Chan */
       2,                      /* strain1Gain */
       0.0,                    /* strain1Zero */
       1,                      /* strain2Chan */
       2,                      /* strain2Gain */
       0.0,                    /* strain2Zero */
       0,                      /* extTach */
       2,                      /* tachChan */
       2,                      /* tachGain */
       0.0,                    /* tachZero */
       0,                      /* uChan */
       0.0,                    /* uZero */
       0xc,                    /* nuConSlot */
       1,                      /* yAxis */
       21333.33333,            /* yCntsPerUnit */
       0.0,                    /* yOffset */
       0.991,                  /* Ktach (V*S/REV) */
};

static sMode Mode = modeOff;
static sCtrlStatus CtrlStatus = 0x0;

static sCmdResult CmdResult = {
        NO_OP,          /* Cmd */
        0,              /* Result */
};

static sFuncPrd FuncPrd = {
        DEFAULT_SAMPLEPRD,
        DEFAULT_CTRLPRD,
        DEFAULT_MONITORPRD,
        DEFAULT_CMDPRD,
        DEFAULT_WATCHDOGPRD,
};

        /* sample period in seconds */
static Flt Ts = DEFAULT_SAMPLEPRD / (Flt) 1000000.0;
        /* control period in seconds */
static Flt Tc = (DEFAULT_SAMPLEPRD * DEFAULT_CTRLPRD) / (Flt) 1000000.0;

static sCtrlParms CtrlParms = {
        200.0,                  /* Kp */
        36.0,                   /* Kd */
        8.0,                    /* Kf1 */
        10.0,                   /* Kf2 */
        0.0,                    /* Kf1d */
        0.0,                    /* Kf2d */
        -2.699,                 /* qfparm1 */
        0.0012,                 /* qfparm2 */
        -5.0085,                /* qfparm3 */
        -0.5106,                /* qfparm4 */
        0.8189,                 /* qfparm5 */
        0.68564,                /* s1ref */
        -1.33259,               /* s2ref */
        0.0,                    /* Ki */
        0.0,                    /* Krd */
        70.0,                   /* Wd */
        10.0,                   /* StrainWd */
        500.0,                  /* VrefWd */
        500.0,                  /* ArefWd */
        0,                      /* Disflx */
        2.0,                    /* Stfrccw */
        -2.0,                   /* Stfrcw */
        0,                      /* Enfric */
        1.0,                    /* Vfrccw */
        1.0,                    /* Vfrcw */
        0,                      /* EnIntegral */

        /****************** Parameters for Neural Net   ***********************/

        0.000001,               /* Kappa */
        2.0,                    /* F */
        20.0,                   /* G */
        0,                      /* addNeural button */

        /*********************************************************************/

};

static sSigLimits SigLimits = {
        0,              /* rLimit */
        0.0,            /* rMax */
        0.0,            /* rMin */
        0,              /* uLimit */
        0.0,            /* uMax */
        0.0,            /* uMin */
};

static sRateLimit RateLimit = {
        0,              /* rateLimit */
        0.0,            /* rateMax */
};

static sFolError FolError = {
        -1,             /* eLimit */
        0.5,            /* eMax */
};

static sWaveParms WaveParms = {
        0.0,    /* Freq */
        0.0,    /* Amp */
        0.0,    /* Offset */
};

static sSigMonitor SigMonitor = {     /*      signal monitor */
        -1,             /* i */
        {       0.0,    /* Sig.r */
                0.0,    /* Sig.y */
                0.0,    /* Sig.u */
                0.0,    /* Sig.v */
                0.0,    /* Sig.qf1 */
                0.0,    /* Sig.qf2 */
                0.0,    /* Sig.dqf1 */
                0.0,    /* Sig.dqf2 */
                0.0,    /* Sig.vs1 */
                0.0,    /* Sig.vs2 */
        },
};

static Int16 CtrlInstalled = 0;         /* true-->controller is installed */
static Int16 CtrlActive = 0;            /* true-->controller is active */
static Int32 WaveSize;                  /* size of waveform table in samples */
static Flt WavePhase;                   /* phase of output waveform */

static Int32 SyncSample;                /* synchronization counter for Sample function */
static Int32 SyncCtrl;                  /* synchronization counter for Ctrl function */
static Int32 SyncMonitor;               /* synchronization counter for Monitor function */
static Int16 Sampled;                   /* false-->first iteration of Sample function */
static Int16 WatchdogTimeout;           /* true-->watchdog timeout */

static Flt y;                           /* measured position at time (ks*Ts) */
static Flt v;                   /* derivative of y or tach measurement at time (ks*Ts) */
static Flt r;                           /* reference signal at time (kc*Tc) */
static Flt rV;                          /* rate limited reference signal at time (kc*Tc) */
static Flt rkp2;                        /* r(kc+2) */
static Flt rkp1;                        /* r(kc+1) */
static Flt rk;                          /* r(kc) */
static Flt rkm1;                        /* r(kc-1) */
static Flt rkm2;                        /* r(kc-2) */
static Flt vrkp1;                       /* reference velocity divided difference at time
(kc+1)*Tc -> vr(kc+1) */
static Flt vrk;                         /*      reference velocity divided difference at time
(kc)*Tc -> vr(kc) */
static Flt vrkm1;                       /*      reference velocity divided difference at time
(kc-1)*Tc -> vr(kc-1) */
static Flt ark;                         /*      reference acceleration divided difference at
time (kc)*Tc -> r(kc) */
static Flt u;                           /*      control signal at time (kc*Tc) */
static Flt uL;                          /*      limited control signal at time (kc*Tc) */
static Flt vs1;                         /*      strain gauge 1 voltage */
static Flt vs2;                         /*      strain gauge 2 voltage */
static Flt qf1;                         /*      deflection 1 at time (ks*Ts) */
static Flt qf2;                         /*      deflection 2 at time (ks*Ts) */
static Flt dqf1;                        /*      derivative of deflection 1 at time (ks*Ts) */
static Flt dqf2;                        /*      derivative of deflection 2 at time (ks*Ts) */

static Flt e;                   /*  position error */
static Flt ekm1;                /*  e(kc-1) */
static Flt ed;                  /*  velocity error */
static Flt eint;                /*  integral of position error */
static Flt fltre;               /*  filtered error */
static Flt absfltre = 0;        /*  Absolute value of filtered error */
static Flt fabsfltre = 0;       /*  Filtered absolute value of filtered error */
static Flt x[N1];               /*  Input vector for Neural Net  */


/*
------------------------------------------------------------------------------------
InstallCtrl

Create the function list and allocate the parameter and signal buffers.  The
controller functions are not activated.  No action is taken if the controller is
already installed.  Returns 0 if installation is successful, -1 otherwise.

Prototype:
LVBool InstallCtrl(Int32 inputQSize, Int32 monitorQSize, Int32 waveSize);
------------------------------------------------------------------------------------
*/
LVBool InstallCtrl(Int32 inputQSize, Int32 monitorQSize, Int32 waveSize)
{
        Int16 i;
        Int32 n;
        sWave waveInit;

        if (CtrlInstalled)     /*      don't do anything if we are already installed */
                return 0;

                /*      create function list */
        CreateTimerFuncList(N_FUNCS);

                /*      allocate buffers */
        CreateBufferList(N_BUFS);

        Bufs.CmdQ = AllocateBuffer(CMDQSIZE);
        Bufs.InputQ = AllocateBuffer(inputQSize * sizeof(sInputQ));
        Bufs.MonitorQ = AllocateBuffer(monitorQSize * sizeof(sMonitorQ));
        Bufs.Wave = AllocateBuffer(waveSize * sizeof(sWave));
        Bufs.Config = AllocateBuffer(sizeof(sConfig));
        Bufs.Parms = AllocateBuffer(sizeof(sParms));

    /************ Allocate Vectors and Matrices for Neural Net *************/

        if(AllocateNN()) {
                RemoveCtrl();
                return -1;
        }

    /***********************************************************************/

        WaveSize = (waveSize > 0) ? waveSize : 0;

                /*      initialize configuration buffer and parameter buffer */
        if (
                WriteFixedBuf(Bufs.Config, 0, sizeof(sConfig), &Config) < (Int32)
sizeof(sConfig)
                || WriteFixedBuf(Bufs.Parms, MODE, sizeof(sMode), &Mode) < (Int32)
sizeof(sMode)
                || WriteFixedBuf(Bufs.Parms, CTRLSTATUS, sizeof(sCtrlStatus), &CtrlStatus)
< (Int32) sizeof(sCtrlStatus)
                || WriteFixedBuf(Bufs.Parms, CMDRESULT, sizeof(sCmdResult), &CmdResult) <
(Int32) sizeof(sCmdResult)
                || WriteFixedBuf(Bufs.Parms, FUNCPRD, sizeof(sFuncPrd), &FuncPrd) < (Int32)
sizeof(sFuncPrd)
                || WriteFixedBuf(Bufs.Parms, CTRLPARMS, sizeof(sCtrlParms), &CtrlParms) <
(Int32) sizeof(sCtrlParms)
                || WriteFixedBuf(Bufs.Parms, SIGLIMITS, sizeof(sSigLimits), &SigLimits) <
(Int32) sizeof(sSigLimits)
                || WriteFixedBuf(Bufs.Parms, RATELIMIT, sizeof(sRateLimit), &RateLimit) <
(Int32) sizeof(sRateLimit)
                || WriteFixedBuf(Bufs.Parms, FOLERROR, sizeof(sFolError), &FolError) <
(Int32) sizeof(sFolError)
```

```
                || WriteFixedBuf(Bufs.Parms, WAVEPARMS, sizeof(sWaveParms), &WaveParms) <
(Int32) sizeof(sWaveParms)
                || WriteFixedBuf(Bufs.Parms, SIGMONITOR, sizeof(sSigMonitor), &SigMonitor)
< (Int32) sizeof(sSigMonitor)
                )
        {
                RemoveCtrl();
                return -1;
        }
                /*      zero waveform buffer */
        waveInit = 0.0;
        for (n = 0; n < WaveSize; n++)
                WriteFixedBuf(Bufs.Wave, n * sizeof(sWave), sizeof(sWave), &waveInit);

        CtrlInstalled = 1;
        return 0;
}

/*
-----------------------------------------------------------------------------------------
RemoveCtrl

Uninstall the controller.

Prototype:
Void RemoveCtrl(Void);
-----------------------------------------------------------------------------------------
*/
Void RemoveCtrl(Void)
{
        DeactivateCtrl();

        DisposeTimerFuncList();
        Bufs.CmdQ = Bufs.InputQ = Bufs.MonitorQ = Bufs.Wave
                = Bufs.Config = Bufs.Parms = -1;
        DisposeBufferList();

    /***************** Free memory used by Neural Net **********************/

    DisposeNN();

    /*********************************************************************/

        CtrlInstalled = 0;

        return;
}

/*
-----------------------------------------------------------------------------------------
ActivateCtrl

Initiate the periodic execution of the controller functions.  No action is taken
if the controller is not installed or it is already active.  The device
configuration parameters are read from the Config buffer and used to initialize the
I/O hardware.  Returns 0 if the controller is successfully activated, -1 otherwise.

Prototype:
LVBool ActivateCtrl(Void);
-----------------------------------------------------------------------------------------
*/
LVBool ActivateCtrl(Void)
{
        TMFuncList flInit;

        if (CtrlInstalled && !CtrlActive) {
                ReadFixedBuf(Bufs.Config, 0, sizeof(sConfig), &Config);
                if (InitMIO16(&Config.MIOConfig) < 0 || InitNuControl(Config.nuConSlot) < 0
                                || ZeroNuConAxisPos(Config.nuConSlot, Config.yAxis) < 0)
                        return -1;

                Mode = modeOff;
                Funcs.Watchdog = InstallTimerFunc(&flInit);  /*      install Watchdog
function */

                SyncSample = SyncCtrl = SyncMonitor = 0;
                Sampled = WatchdogTimeout = 0;
                y = v = r = rV = rkp2 = rkp1 = rk =rkm1 = rkm2 = u = uL = qf1 = qf2 = dqf1
= dqf2 = 0.0;
                vs1 = vs2 = vrkp1 = vrk = vrkm1 = ark = ekm1 = eint = 0.0;

                CtrlActive = 1;

                if (Funcs.Sample == -1 || Funcs.Ctrl == -1 || Funcs.Monitor == -1
                        || Funcs.Cmd == -1 || Funcs.Watchdog == -1) {
                        DeactivateCtrl();
                        return -1;
                }
                ActivateTimerFuncs(N_FUNCS, (Int16 *) &Funcs);
        }

        if (CtrlActive)
                return 0;
        else
                return -1;
}

/*
-------------------------------------------------------------------------------------
DeactivateCtrl

Inhibit the periodic execution of the controller functions.  A stop command is issued
before the controller is deactivated.  No action is performed if the controller is
not active.

Prototype:
Void DeactivateCtrl(Void);
-------------------------------------------------------------------------------------
*/
Void DeactivateCtrl(Void)
{
        if (CtrlActive) {
                EStop();
                DeactivateTimerFuncs(N_FUNCS, (Int16 *) &Funcs);
                RemoveTimerFunc(Funcs.Watchdog);
                RemoveTimerFunc(Funcs.Cmd);
                RemoveTimerFunc(Funcs.Monitor);
                RemoveTimerFunc(Funcs.Ctrl);
                RemoveTimerFunc(Funcs.Sample);
                Funcs.Sample = Funcs.Ctrl = Funcs.Monitor = Funcs.Cmd = Funcs.Watchdog = -
1;
                CtrlActive = 0;
        }

        return;
}

/*
-------------------------------------------------------------------------------------
EStop

Perform an emergency stop.  The controller is switched off and the control signal is
immediately set to zero.

Prototype:
Void EStop(Void);
-------------------------------------------------------------------------------------
*/
Void EStop(Void)
{
        Uint16 sReg;

        sReg = BlockInterrupts();     /*      make sure we aren't interrupted */
```

```
                fl->period = FuncPrd.Sample;

        yl = y;
        ReadNuConAxisPos(Config.nuConSlot, Config.yAxis, &posCnt);
        y = ScaleNuConAxisPos(posCnt, Config.yCntsPerUnit, Config.yOffset);

        SetMIO16ADChannel(&Config.MIOConfig, Config.strain1Chan, Config.strain1Gain);
        vs1 = - Config.strain1Zero
                        + ScaleMIO16ADC(&Config.MIOConfig, ReadMIO16ADC(&Config.MIOConfig),
Config.strain1Gain);

        SetMIO16ADChannel(&Config.MIOConfig, Config.strain2Chan, Config.strain2Gain);
        vs2 = - Config.strain2Zero
                        + ScaleMIO16ADC(&Config.MIOConfig, ReadMIO16ADC(&Config.MIOConfig),
Config.strain2Gain);

/*
%%%%%%%%%%%%%%%%%%%%  Convert Strain volts to deflection magnitudes
*/
        qf1l = qf1;
        qf2l = qf2;
        qf1 = CtrlParms.qfparm1 * (vs1 - CtrlParms.s1ref) / ((Flt) 1.0 + CtrlParms.qfparm2
* (vs1 - CtrlParms.s1ref))
                        + CtrlParms.qfparm3 * (vs2 - CtrlParms.s2ref) / ((Flt)1.0 +
CtrlParms.qfparm2 * (vs2 - CtrlParms.s2ref));
        qf2 = CtrlParms.qfparm4 * (vs1 - CtrlParms.s1ref) / ((Flt) 1.0 + CtrlParms.qfparm2
* (vs1 - CtrlParms.s1ref))
                        + CtrlParms.qfparm5 * (vs2 - CtrlParms.s2ref) / ((Flt) 1.0 +
CtrlParms.qfparm2 * (vs2 - CtrlParms.s2ref));

        if (!Sampled) {
                yl = y;
                qf1l = qf1;
                qf2l = qf2;
                Sampled = 1;
        }

        if (Config.extTach < 0) {
                SetMIO16ADChannel(&Config.MIOConfig, Config.tachChan, Config.tachGain);
                v = Config.Ktach * (- Config.tachZero
                        + ScaleMIO16ADC(&Config.MIOConfig, ReadMIO16ADC(&Config.MIOConfig),
Config.tachGain));
        }
        else {
                if (CtrlParms.Wd) {
                        Flt a = (Flt) 1.0 / ((Flt) 1.0 + CtrlParms.Wd * Ts);

                        v = a * v + CtrlParms.Wd * a * (y - yl);
                }
                else
                        v = 0.0;
        }

        if (CtrlParms.StrainWd) {
                Flt a = (Flt) 1.0 / ((Flt) 1.0 + CtrlParms.StrainWd * Ts);

                dqf1 = a * dqf1 + CtrlParms.StrainWd * a * (qf1 - qf1l);
                dqf2 = a * dqf2 + CtrlParms.StrainWd * a * (qf2 - qf2l);
        }
        else {
                dqf1 = 0.0;
                dqf2 = 0.0;
        }

        return;
}

/*
-----------------------------------------------------------------------------------
Ctrl

Prototype:
```

```
Void Ctrl(TMFuncList *fl);
--------------------------------------------------------------------------------------
*/
Void Ctrl(TMFuncList *fl)
{
        Flt rL;                 /*      limited reference signal at time (kc+2)*Tc) */
        Flt uNN = 0.0;          /*      neural net control contribution */
        Flt lamda;              /*  parameter for filtered error calculation */
        Flt Friction;           /*  simulated friction term */

        Flt Vo = 0.05;       /*  threshold velocity for friction model */

        /*      update period only when synchronized with Cmd function */
        if (!SyncCtrl--)
                fl->period = FuncPrd.Sample * FuncPrd.Ctrl;

        switch (Mode) {
                case modeOff:
                default:
                        u = uL = 0.0;
                        r = rL = 0.0;
                        break;
                case modeAuto:
                case modePrdAuto:
                        if (Mode == modeAuto) {
                                /*      read reference signal from input queue */
                                if (ReadBuffer(Bufs.InputQ, sizeof(sInputQ), &r) < (Int32)
sizeof(sInputQ))
                                        SetBufEmpty(Bufs.InputQ);
                        }
                        else {  /*      read reference signal from waveform buffer */
                                if (WaveSize > 0) {
                                        sWave wave1, wave2;
                                        Flt waveIndex;
                                        Int32 n1, n2;

                                        if (WavePhase >= (Flt) 1.0)
                                                WavePhase -= (Flt) ((Int32) WavePhase);
                                        n1 = (Int32) (waveIndex = WavePhase * (Flt)
WaveSize);
                                        if ((n2 = n1 + 1) == WaveSize)
                                                n2 = 0;
                                        ReadFixedBuf(Bufs.Wave, n1 * sizeof(sWave),
sizeof(sWave), &wave1);
                                        ReadFixedBuf(Bufs.Wave, n2 * sizeof(sWave),
sizeof(sWave), &wave2);
                                        r = WaveParms.Offset +
                                                WaveParms.Amp * (wave1 + (waveIndex - (Flt)
n1) * (wave2 - wave1));
                                        WavePhase += WaveParms.Freq * Tc;
                                }
                        }

                        /*      limit the reference signal */
                        if (SigLimits.rLimit < 0)
                                rL = (r < SigLimits.rMin ? SigLimits.rMin : (r <
SigLimits.rMax ? r : SigLimits.rMax));
                        else
                                rL = r;

                        /*      limit the rate of change of the reference signal */
                        if (RateLimit.rateLimit < 0) {
                                Flt dr = RateLimit.rateMax * Tc;

                                rV = ((rL - rV < - dr) ? rV - dr : ((rL - rV < dr) ? rL : rV
+ dr));
                        }
                        else
                                rV = rL;

/*
----------------
```

```
        Better age the old values of the limited reference signal
-----------------
*/
                        rkm2 = rkm1; rkm1 = rk; rk = rkp1; rkp1 = rkp2; rkp2 = rV;
                        vrkm1 = vrk; vrk = vrkp1;

                        /*      quantize the reference signal */
                        rL = rk;
                        QuantizeNuConAxisPos(&rL, Config.yCntsPerUnit, Config.yOffset);

                        e = rL - y;

/* MODIFIED to increase safety of operation:
the emergency stop is produced when qf1 and/or qf2 are big enough too */

                        if (FolError.eLimit < 0) {
                                if ((e > (Flt) 0.0 && e > FolError.eMax) || (e < (Flt) 0.0
&& -e > FolError.eMax) || ((qf1*qf1 + qf2*qf2) > 100*FolError.eMax*FolError.eMax)) {
                                        EStop();        /*      following error */
                                        CtrlStatus |= CS_FOLERROR;
                                        WriteFixedBuf(Bufs.Parms, CTRLSTATUS,
sizeof(sCtrlStatus), &CtrlStatus);
                                        break;
                                }
                        }

/* ------------------------------------------------------------------------
        Insert here the computation of reference velocity and acceleration
   ------------------------------------------------------------------------*/

                        if (CtrlParms.VrefWd) {
                                Flt a1 = (Flt) 1.0 / ((Flt) 1.0 + (Flt) 2.0 *
CtrlParms.VrefWd * Tc);

                                vrkp1 = a1 * vrkp1 + a1 * CtrlParms.VrefWd * (rkp2 -  rk);
                        }
                        else  {
                                vrkp1 = 0.0;
                        }

                        if (CtrlParms.ArefWd) {
                                Flt a1 = (Flt)1.0 / ((Flt)1.0 + (Flt)2.0 * CtrlParms.ArefWd
* Tc);

                                ark = a1 * ark + a1 * CtrlParms.ArefWd * (vrkp1-vrkm1);
                        }
                        else  {
                                ark = 0.0;
                        }

                        ed = vrk - v;           /* edot */


                        u = CtrlParms.Kp * e + CtrlParms.Kd * ed;    /* PD control */


                eint += Tc * (e + ekm1) / 2;    /* update integral */
                ekm1 = e;

                if (eint < -FolError.eMax)
                   eint = -FolError.eMax;
                if (eint > FolError.eMax)
                   eint = FolError.eMax;        /* limit integral */


                        if (CtrlParms.EnIntegral < 0)
                                u += CtrlParms.Ki * eint;     /* include integral control */


                        if (CtrlParms.Disflx >= 0) {
                                u += CtrlParms.Kf1 * qf1 + CtrlParms.Kf2 * qf2 +
CtrlParms.Kf1d * dqf1 + CtrlParms.Kf2d * dqf2;
```

```
                        }
/***********************  Apply the neural net term  **********************/

                        if (CtrlParms.addNeural < 0) {
                         Flt a = (Flt) 2.0 / (CtrlParms.StrainWd * Ts);

                        x[NN_INPUT_e] = e;               /*  evaluate input vector  */
                        x[NN_INPUT_ed] = ed;
                        x[NN_INPUT_r] = r;
                        x[NN_INPUT_vrk] = vrk;
                        x[NN_INPUT_ark] = ark;
                        if (CtrlParms.Kd != 0)
                               lamda = CtrlParms.Kp / CtrlParms.Kd;
                        if (CtrlParms.Kd == 0 || lamda > 100.0)
                               lamda = 100.0;
                        fltre = ed + lamda * e;         /* calculate filtered error */



                        uNN = NeuralNetCtrl(x , fltre , Ts , &CtrlParms);
                        u += uNN;
                        }

/**************************************************************************/


/***********************  Simulate extra friction *************************/

                        if (CtrlParms.Enfric < 0) {
                               if (v*v <= Vo*Vo) {
                                  if (u > (Flt) 0.0) {
                                     Friction = 2 * CtrlParms.Vfrccw * Vo +
CtrlParms.Stfrccw;
                                 u -= Friction;
                                     if (u < (Flt) 0.0)
                                            u = 0.0;
                                  }
                                  if (u < (Flt) 0.0) {
                                     Friction = -2 * CtrlParms.Vfrcw * Vo +
CtrlParms.Stfrcw;
                                        u -= Friction;
                                     if (u > (Flt) 0.0)
                                            u = 0.0;
                                  }
                               }
                               if (v > Vo) {
                                   Friction = CtrlParms.Vfrccw * v + CtrlParms.Stfrccw;
                               u -= Friction;
                               }
                               if (v < -Vo) {
                                   Friction = CtrlParms.Vfrcw * v + CtrlParms.Stfrcw;
                                      u -= Friction;
                               }
                        }

/**************************************************************************/


/*
~~~~~~~~~~~~~~~    Limit and Convert the Control Signal ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*/
                        u += Config.uZero;
                        if (SigLimits.uLimit < 0)
                               uL = (u < SigLimits.uMin ? SigLimits.uMin : (u <
SigLimits.uMax ? u : SigLimits.uMax));
                        else
                               uL = u;
                        u -= Config.uZero;
                        uL = - Config.uZero + LimitMIO16DAC(&Config.MIOConfig, uL);

                        break;
```

```
                case modeManual:
                case modePrdManual:
                        if (Mode == modeManual) {
                                /*      read control signal from input queue */
                                if (ReadBuffer(Bufs.InputQ, sizeof(sInputQ), &u) < (Int32)
sizeof(sInputQ))
                                        SetBufEmpty(Bufs.InputQ);
                        }
                        else {  /*      read control signal from waveform buffer */
                                if (WaveSize > 0) {
                                        sWave wave1, wave2;
                                        Flt waveIndex;
                                        Int32 n1, n2;

                                        if (WavePhase >= (Flt) 1.0)
                                                WavePhase -= (Flt) ((Int32) WavePhase);
                                        n1 = (Int32) (waveIndex = WavePhase * (Flt)
WaveSize);
                                        if ((n2 = n1 + 1) == WaveSize)
                                                n2 = 0;
                                        ReadFixedBuf(Bufs.Wave, n1 * sizeof(sWave),
sizeof(sWave), &wave1);
                                        ReadFixedBuf(Bufs.Wave, n2 * sizeof(sWave),
sizeof(sWave), &wave2);
                                        u = WaveParms.Offset +
                                                WaveParms.Amp * (wave1 + (waveIndex - (Flt)
n1) * (wave2 - wave1));
                                        WavePhase += WaveParms.Freq * Tc;
                                }
                        }

                        u += Config.uZero;
                        if (SigLimits.uLimit < 0)
                                uL = (u < SigLimits.uMin ? SigLimits.uMin : (u <
SigLimits.uMax ? u : SigLimits.uMax));
                        else
                                uL = u;
                        u -= Config.uZero;
                        uL = - Config.uZero + LimitMIO16DAC(&Config.MIOConfig, uL);

                        r = rL = 0.0;
                        break;
        }

        uL += Config.uZero;
        WriteMIO16DAC(&Config.MIOConfig, Config.uChan, ScaleMIO16DAC(&Config.MIOConfig,
&uL));
        uL -= Config.uZero;

        SigMonitor.Sig.r = rL;
        SigMonitor.Sig.y = y;
        SigMonitor.Sig.u = uL;
        SigMonitor.Sig.v = v;
        SigMonitor.Sig.qf1 = qf1;
        SigMonitor.Sig.qf2 = qf2;
        SigMonitor.Sig.dqf1 = dqf1;
        SigMonitor.Sig.dqf2 = dqf2;
        SigMonitor.Sig.vs1 = vs1;
        SigMonitor.Sig.vs2 = vs2;

        return;
}

/*
-------------------------------------------------------------------------------------
Monitor

Prototype:
Void Monitor(TMFuncList *fl);
-------------------------------------------------------------------------------------
*/
Void Monitor(TMFuncList *fl)
```

```
{
        /*      update period only when synchronized with Cmd function */
        if (!SyncMonitor--)
                fl->period = FuncPrd.Sample * FuncPrd.Ctrl * FuncPrd.Monitor;

                /*      write signals to parameter buffer */
        if (++SigMonitor.i < 0)
                SigMonitor.i = 0;
        WriteFixedBuf(Bufs.Parms, SIGMONITOR, sizeof(sSigMonitor), &SigMonitor);

                /*      write signals to signal monitor buffer */
        if (WriteBuffer(Bufs.MonitorQ, sizeof(sMonitorQ), &SigMonitor.Sig) < (Int32)
sizeof(sMonitorQ))
                SetBufDataLost(Bufs.MonitorQ);

        /* debug */
        CmdResult.Result = sizeof(sCtrlParms);
        WriteFixedBuf(Bufs.Parms, CMDRESULT, sizeof(sCmdResult), &CmdResult);

        return;
}

/*
-------------------------------------------------------------------------------------
Cmd

Command result codes returned in CmdResult.Result:    0 --> no error

        -1 --> could not complete

Prototype:
Void Cmd(TMFuncList *fl);
-------------------------------------------------------------------------------------
*/
Void Cmd(TMFuncList *fl)
{
        Int32 nBytes;
        sCmds CmdBuf;

        fl->period = FuncPrd.Sample * FuncPrd.Ctrl * FuncPrd.Monitor * FuncPrd.Cmd;
        Ts = FuncPrd.Sample / (Flt) 1000000.0;
        Tc = (FuncPrd.Sample * FuncPrd.Ctrl) / (Flt) 1000000.0;
        SyncSample = FuncPrd.Ctrl * FuncPrd.Monitor * FuncPrd.Cmd - 1;
        SyncCtrl = FuncPrd.Monitor * FuncPrd.Cmd - 1;
        SyncMonitor = FuncPrd.Cmd - 1;

        /*      command processor */
        if ((nBytes = CheckBuffer(Bufs.CmdQ, cbUsed)) < (Int32) sizeof(CmdBuf.Size)) {
                if (nBytes > 0)
                        FlushBuffers(1, &Bufs.CmdQ);  /* incomplete command packet */
                return;
        }
        ReadBuffer(Bufs.CmdQ, sizeof(CmdBuf.Size), &CmdBuf.Size);   /*      read command
size */
        nBytes -= sizeof(CmdBuf.Size);
        if (CmdBuf.Size < 0) {
                FlushBuffers(1, &Bufs.CmdQ);  /* garbled command packet, flush entire
buffer */
                return;
        }
        if (CmdBuf.Size >= (Int32) sizeof(CmdBuf.Cmd)) {
                ReadBuffer(Bufs.CmdQ, sizeof(CmdBuf.Cmd), &CmdBuf.Cmd);     /*      read
command opcode */
                nBytes -= sizeof(CmdBuf.Cmd);
                CmdBuf.Size -= sizeof(CmdBuf.Cmd);
        }
        else {
                ReadBuffer(Bufs.CmdQ, CmdBuf.Size, 0);       /*      incomplete command
packet */
                return;
        }
        WatchdogTimeout = 0;            /* clear timeout flag */

        CmdResult.Cmd = CmdBuf.Cmd;
        CmdResult.Result = 0;
        switch (CmdBuf.Cmd) {
                case WR_MODE:
                        if (nBytes >= (Int32) sizeof(sMode) && CmdBuf.Size >= (Int32)
sizeof(sMode)) {
                                ReadBuffer(Bufs.CmdQ, sizeof(sMode), &CmdBuf.Data);
                                CmdBuf.Size -= sizeof(sMode);
                                switch (CmdBuf.Data.Mode) {
                                        case modeOff:
                                                CtrlStatus = 0x0;
                                                WriteFixedBuf(Bufs.Parms, CTRLSTATUS,
sizeof(sCtrlStatus), &CtrlStatus);
                                                if (Mode != modeOff) {
                                                        Mode = modeOff;
                                                        WriteFixedBuf(Bufs.Parms, MODE,
sizeof(sMode), &Mode);
                                                }
                                                break;
                                        case modeAuto:
                                                CtrlStatus = 0x0;
                                                WriteFixedBuf(Bufs.Parms, CTRLSTATUS,
sizeof(sCtrlStatus), &CtrlStatus);
                                                if (Mode != modeAuto) {
                                                        FlushBuffers(1, &Bufs.InputQ);
                                                        r = rV = rkp2 = rkp1 = rk = rkm1 =
rkm2 = y;
                                                        vrk = vrkp1 = vrkm1 = ark = 0.0;
                                                        Mode = modeAuto;
                                                        WriteFixedBuf(Bufs.Parms, MODE,
sizeof(sMode), &Mode);
                                                }
                                                break;
                                        case modePrdAuto:
                                                CtrlStatus = 0x0;
                                                WriteFixedBuf(Bufs.Parms, CTRLSTATUS,
sizeof(sCtrlStatus), &CtrlStatus);
                                                if (Mode != modePrdAuto) {
                                                        WavePhase = 0.0;
                                                        WaveParms.Amp = 0.0;
                                                        WaveParms.Offset = y;
                                                        WriteFixedBuf(Bufs.Parms, WAVEPARMS,
sizeof(sWaveParms), &WaveParms);
                                                        r = rV = rkp2 = rkp1 = rk = rkm1 =
rkm2 = y;
                                                        vrk = vrkp1 = vrkm1 = ark = 0.0;
                                                   /*   ui = uL;                initialize
integrator for bumpless transfer */
                                                        Mode = modePrdAuto;
                                                        WriteFixedBuf(Bufs.Parms, MODE,
sizeof(sMode), &Mode);
                                                }
                                                break;
                                        case modeManual:
                                                CtrlStatus = 0x0;
                                                WriteFixedBuf(Bufs.Parms, CTRLSTATUS,
sizeof(sCtrlStatus), &CtrlStatus);
                                                if (Mode != modeManual) {
                                                        FlushBuffers(1, &Bufs.InputQ);
                                                        Mode = modeManual;
                                                        WriteFixedBuf(Bufs.Parms, MODE,
sizeof(sMode), &Mode);
                                                }
                                                break;
                                        case modePrdManual:
                                                CtrlStatus = 0x0;
                                                WriteFixedBuf(Bufs.Parms, CTRLSTATUS,
sizeof(sCtrlStatus), &CtrlStatus);
                                                if (Mode != modePrdManual) {
                                                        WavePhase = 0.0;
                                                WaveParms.Amp = 0.0;
                                                WaveParms.Offset = uL;
                                                WriteFixedBuf(Bufs.Parms, WAVEPARMS,
sizeof(sWaveParms), &WaveParms);

                                                Mode = modePrdManual;
                                                WriteFixedBuf(Bufs.Parms, MODE,

sizeof(sMode), &Mode);
                                        }
                                        break;
                                default:
                                        CmdResult.Result = -1; /*      unrecognized
mode */
                                        break;
                        }
                }
                else
                        CmdResult.Result = -1; /*      incomplete command packet */
                break;
        case WR_FUNCPRD:
                if (nBytes >= (Int32) sizeof(sFuncPrd) && CmdBuf.Size >= (Int32)
sizeof(sFuncPrd)) {
                        ReadBuffer(Bufs.CmdQ, sizeof(sFuncPrd), &CmdBuf.Data);
                        CmdBuf.Size -= sizeof(sFuncPrd);
                        if (CmdBuf.Data.FuncPrd.Sample >= MIN_PERIOD &&
CmdBuf.Data.FuncPrd.Ctrl > 0
                                && CmdBuf.Data.FuncPrd.Monitor > 0 &&
CmdBuf.Data.FuncPrd.Cmd > 0
                                && CmdBuf.Data.FuncPrd.Watchdog >= MIN_PERIOD)
                        {
                                FuncPrd.Sample = (Int32) (CmdBuf.Data.FuncPrd.Sample
/ 20) * 20;
                                FuncPrd.Ctrl = CmdBuf.Data.FuncPrd.Ctrl;
                                FuncPrd.Monitor = CmdBuf.Data.FuncPrd.Monitor;
                                FuncPrd.Cmd = CmdBuf.Data.FuncPrd.Cmd;
                                FuncPrd.Watchdog = (Int32)
(CmdBuf.Data.FuncPrd.Watchdog / 20) * 20;
                                WriteFixedBuf(Bufs.Parms, FUNCPRD, sizeof(sFuncPrd),
&FuncPrd);
                        }
                        else
                                CmdResult.Result = -1; /*      bad function periods
*/
                }
                else
                        CmdResult.Result = -1; /*      incomplete command packet */
                break;
        case WR_CTRLPARMS:
                if (nBytes >= (Int32) sizeof(sCtrlParms) && CmdBuf.Size >= (Int32)
sizeof(sCtrlParms)) {
                        ReadBuffer(Bufs.CmdQ, sizeof(sCtrlParms), &CmdBuf.Data);
                        CmdBuf.Size -= sizeof(sCtrlParms);
                        CtrlParms = CmdBuf.Data.CtrlParms;
                        WriteFixedBuf(Bufs.Parms, CTRLPARMS, sizeof(sCtrlParms),
&CtrlParms);
                }
                else
                        CmdResult.Result = -1; /*      incomplete command packet */
                break;
        case WR_SIGLIMITS:
                if (nBytes >= (Int32) sizeof(sSigLimits) && CmdBuf.Size >= (Int32)
sizeof(sSigLimits)) {
                        ReadBuffer(Bufs.CmdQ, sizeof(sSigLimits), &CmdBuf.Data);
                        CmdBuf.Size -= sizeof(sSigLimits);
                        if ((CmdBuf.Data.SigLimits.rLimit >= 0 ||
CmdBuf.Data.SigLimits.rMin <= CmdBuf.Data.SigLimits.rMax)
                                && (CmdBuf.Data.SigLimits.uLimit >= 0 ||
CmdBuf.Data.SigLimits.uMin <= Config.uZero
                                        && CmdBuf.Data.SigLimits.uMax >=
Config.uZero))
                        {
                                SigLimits = CmdBuf.Data.SigLimits;
```

```
					WriteFixedBuf(Bufs.Parms, SIGLIMITS,
sizeof(sSigLimits), &SigLimits);
				}
				else
					CmdResult.Result = -1; /*	bad limits */
			}
			else
				CmdResult.Result = -1; /*	incomplete command packet */
			break;
		case WR_RATELIMIT:
			if (nBytes >= (Int32) sizeof(sRateLimit) && CmdBuf.Size >= (Int32)
sizeof(sRateLimit)) {
				ReadBuffer(Bufs.CmdQ, sizeof(sRateLimit), &CmdBuf.Data);
				CmdBuf.Size -= sizeof(sRateLimit);
				if (CmdBuf.Data.RateLimit.rateLimit >= 0 ||
CmdBuf.Data.RateLimit.rateMax >= (Flt) 0.0) {
					RateLimit = CmdBuf.Data.RateLimit;
					WriteFixedBuf(Bufs.Parms, RATELIMIT,
sizeof(sRateLimit), &RateLimit);
				}
				else
					CmdResult.Result = -1; /*	bad limit */
			}
			else
				CmdResult.Result = -1; /*	incomplete command packet */
			break;
		case WR_FOLERROR:
			if (nBytes >= (Int32) sizeof(sFolError) && CmdBuf.Size >= (Int32)
sizeof(sFolError)) {
				ReadBuffer(Bufs.CmdQ, sizeof(sFolError), &CmdBuf.Data);
				CmdBuf.Size -= sizeof(sFolError);
				if (CmdBuf.Data.FolError.eLimit >= 0 ||
CmdBuf.Data.FolError.eMax >= (Flt) 0.0) {
					FolError = CmdBuf.Data.FolError;
					WriteFixedBuf(Bufs.Parms, FOLERROR,
sizeof(sFolError), &FolError);
				}
				else
					CmdResult.Result = -1; /*	bad following error */
			}
			else
				CmdResult.Result = -1; /*	incomplete command packet */
			break;
		case WR_WAVEPARMS:
			if (nBytes >= (Int32) sizeof(sWaveParms) && CmdBuf.Size >= (Int32)
sizeof(sWaveParms)) {
				ReadBuffer(Bufs.CmdQ, sizeof(sWaveParms), &CmdBuf.Data);
				CmdBuf.Size -= sizeof(sWaveParms);
				if (CmdBuf.Data.WaveParms.Freq >= (Flt) 0.0) {
					WaveParms = CmdBuf.Data.WaveParms;
					WriteFixedBuf(Bufs.Parms, WAVEPARMS,
sizeof(sWaveParms), &WaveParms);
				}
				else
					CmdResult.Result = -1; /*	bad wave parameters */
			}
			else
				CmdResult.Result = -1; /*	incomplete command packet */
			break;
		case NO_OP:
			break;
		default:
			CmdResult.Result = -1; /*	unrecognized command */
			break;
	}
	if (CmdBuf.Size > 0)
		ReadBuffer(Bufs.CmdQ, CmdBuf.Size, 0);		/*	discard unused bytes
in command packet */
	WriteFixedBuf(Bufs.Parms, CMDRESULT, sizeof(sCmdResult), &CmdResult);

	return;
}
```

```
/*
------------------------------------------------------------------------------------
Watchdog

Prototype:
Void Watchdog(TMFuncList *fl);
------------------------------------------------------------------------------------
*/
Void Watchdog(TMFuncList *fl)
{
        fl->period = FuncPrd.Watchdog;


        if (WatchdogTimeout) {
                EStop();
                CtrlStatus |= CS_TIMEOUT;
                FuncPrd.Sample = DEFAULT_SAMPLEPRD;
                FuncPrd.Ctrl = DEFAULT_CTRLPRD;
                FuncPrd.Monitor = DEFAULT_MONITORPRD;
                FuncPrd.Cmd = DEFAULT_CMDPRD;
                FuncPrd.Watchdog = DEFAULT_WATCHDOGPRD;
                Ts = DEFAULT_SAMPLEPRD / (Flt) 1000000.0;
                Tc = (DEFAULT_SAMPLEPRD * DEFAULT_CTRLPRD) / (Flt) 1000000.0;
                WriteFixedBuf(Bufs.Parms, CTRLSTATUS, sizeof(sCtrlStatus), &CtrlStatus);
                WriteFixedBuf(Bufs.Parms, FUNCPRD, sizeof(sFuncPrd), &FuncPrd);
        }

        WatchdogTimeout = 1;

        return;
}
```

```
/*
-------------------------------------------------------------------------------------
NeuralNet.c

Neural Net Control for Flexible Link.

Luis Gutierrez, 4/9/96
-------------------------------------------------------------------------------------
*/


/*      Include Files */
#include "NNFlexLink.h"

#if __option(a4_globals)
#include "extcode2.h"
#else
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#endif



/*************** Vectors and Matrices for Neural Net   *****************/

static Flt *H;                  /*  Input to Hidden layer */
static Flt *Sigma;              /*  Hidden layer output */
static Flt *SigmaPrime;         /*  Derivatives of hidden layer outputs */
static Flt *W;                  /*  Output layer weights */
static Flt *DeltaW;             /*  Delta of output layer weights */
static Flt *V;                  /*  Input layer weights */
static Flt *DeltaV;             /*  Delta of input layer weights */

/**********************************************************************/



/*
-------------------------------------------------------------------------------------
AllocateNN

Allocate vectors and matrices for the Neural Net. Initializes weight matrices to zero.
Returns 0 if allocation is successful, -1 otherwise.

Prototype:
Int16 AllocateNN(Void);
-------------------------------------------------------------------------------------
*/

Int16 AllocateNN(Void)
{
        Int16 i;

        H = (Flt *) DSNewPtr(N2*sizeof(Flt));
        Sigma = (Flt *) DSNewPtr(N2*sizeof(Flt));
        SigmaPrime = (Flt *) DSNewPtr(N2*sizeof(Flt));
        W = (Flt *) DSNewPtr((N2+1)*sizeof(Flt));
        DeltaW = (Flt *) DSNewPtr((N2+1)*sizeof(Flt));
        V = (Flt *) DSNewPtr((N1+1)*N2*sizeof(Flt));
        DeltaV = (Flt *) DSNewPtr((N1+1)*N2*sizeof(Flt));

        if (!H || !Sigma || !SigmaPrime || !W || !DeltaW || !V|| !DeltaV) {
                DisposeNN();
                return 1;
        }

    /*  initializes weights to zero  */
        for (i=0 ; i < (N2+1) ; i++) {
            W[i] = 0;
            DeltaW[i] = 0;
            }
        for (i=0 ; i < (N1+1)*N2 ; i++) {
            V[i] = 0;
            DeltaV[i] = 0;
            }

        return 0;
}


/*
-------------------------------------------------------------------------------------
DisposeNN

Frees memory used by vectors and matrices for the Neural Net.

Prototype:
Void DisposeNN(Void);
-------------------------------------------------------------------------------------
*/

Void DisposeNN(Void)
{
        if (H)
                DSDisposePtr(H);
        if (Sigma)
                DSDisposePtr(Sigma);
        if (SigmaPrime)
                DSDisposePtr(SigmaPrime);
        if (W)
                DSDisposePtr(W);
        if (DeltaW)
                DSDisposePtr(DeltaW);
        if (V)
                DSDisposePtr(V);
        if (DeltaV)
                DSDisposePtr(DeltaV);

        H=Sigma=SigmaPrime=W=DeltaW=V=DeltaV=0;

        return;
}


/*
-------------------------------------------------------------------------------------
NeuralNetCtrl

Neural net controller. Returns the output of the neural net.

Prototype:
Flt NeuralNetCtrl(Flt x[] , Flt fltre , Flt samplePrd, const sCtrlParms *ctrlParms);
-------------------------------------------------------------------------------------
*/

Flt NeuralNetCtrl(Flt x[] , Flt fltre , Flt samplePrd, const sCtrlParms *ctrlParms)
{
    Int16 i , j;
    Flt KappaSgnFltre , Output , Ts_F_Fltre , Ts_G_Fltre, Alpha=1;

    /*  Calculate H, Sigma , and SigmaPrime  */
    for (j=0;j<N2;j++) {
        H[j] = V[j];
        for (i=1;i<=N1;i++)
            H[j] += V[i*N2+j] * x[i-1];
        Sigma[j] = 1.0 / (1.0 + exp(-(j+1)*Alpha*H[j]));
        SigmaPrime[j] = (j+1) * Alpha * Sigma[j] * (1 - Sigma[j]);
        }

    /*  Calculate Neural Net Output  */
```

```
     Output = W[0];
     for (i=1;i<=N2;i++)
          Output += W[i] * Sigma[i-1];


         /*  Update rules for weight matrices V and W  */

              /*  Consider sign of filtered error  */
         KappaSgnFltre = (fltre >= 0) ? ctrlParms->Kappa : -ctrlParms->Kappa;

          /*  Optimize operations  */
     Ts_G_Fltre = samplePrd * ctrlParms->G * fltre / 2;
     Ts_F_Fltre = samplePrd * ctrlParms->F * fltre / 2;

              /*  Update V  */
         for (j=0;j<N2;j++) {
              V[j] += DeltaV[j];
              DeltaV[j] = Ts_G_Fltre * (W[j+1] * SigmaPrime[j] - KappaSgnFltre * V[j]);
              V[j] += DeltaV[j];
              for (i=1;i<=N1;i++) {
                   V[i*N2+j] += DeltaV[i*N2+j];
                   DeltaV[i*N2+j] = Ts_G_Fltre * (x[i-1] * W[j+1] * SigmaPrime[j] -
KappaSgnFltre * V[i*N2+j]);
                   V[i*N2+j] += DeltaV[i*N2+j];
                   }
              }

          /*  Update W  */
     W[0] += DeltaW[0];
     DeltaW[0] = Ts_F_Fltre * (1 - KappaSgnFltre * W[0]);
     W[0] += DeltaW[0];
     for (i=0;i<N2;i++) {
          W[i+1] += DeltaW[i+1];
          DeltaW[i+1] = Ts_F_Fltre * (Sigma[i] - SigmaPrime[i] * H[i] - KappaSgnFltre *
W[i+1]);
          W[i+1] += DeltaW[i+1];
              }
         return Output;
}
```

REFERENCES

[1] P. J. Antsaklis, Guest Editor of Special Issue on Neural Networks for Control Systems, IEEE Control systems Magazine, vol. 10, no. 3, Apr. 1990.

[2] H. Asada and Z.-D. Ma and H. Tokumaru, "Inverse dynamics of flexible robot arms: modeling and computation for trajectory control," *J. Dynam. Systems, Measurement and Control*, vol. 112, pp. 177-185, Jun. 1990.

[3] W.J. Book, "Modeling, design, and control of flexible manipulator arms: a tutorial review," in *Proc. 29th IEEE Conf. Decision and Control*, Dec. 1990, pp. 500-506.

[4] F.-C. Chen and C.-C. Liu, ``Adaptively controlling nonlinear continuous-time systems using multilayer neural networks," *IEEE Trans. Automat.Control*, vol. 39, no. 6, pp. 1306-1310, 1994.

[5] S. Centikunt, B. Siciliano, and W. J. Book, "Symbolic modeling and dynamic analysis of flexible manipulators," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Oct. 1986, pp.798-803.

[6] A. De Luca, and B. Siciliano, "Closed-form dynamic model of planar multilink lightweight robots," *IEEE Trans. on Systems Man and Cybernetics*, vol. 21, no. 4, pp. 826-839, Jul./Aug. 1991.

[7] D. J. Gorman, *Free Vibration Analysis of Beams and Shafts*. New York, NY: John Wiley, 1975.

[8] C. M. Harris, editor, *Shock and Vibration Handbook*. New York, NY: McGraw-Hill, 1988.

[9] G. G. Hastings and W. J. Book, "A linear dynamic model for flexible robotic manipulators," *IEEE. Control Systems Magazine*, vol. 7, no. 1, pp. 61-64, 1987.

[10] G. G. Hastings and W. J. Book, "Verification of a linear dynamic model for flexible robotic manipulators," in *Proc. IEEE Int. Conf. Robotics and Automation*, Apr. 1986, pp. 1024-1029.

[11] S. Haykin, *Neural Networks*. New York, NY: IEEE Press and Macmillan, 1994.

[12] H. Kanoh, S. Tzafestas, H. G. Lee, and J. Kalat, "Modeling and control of flexible robot arms," in *Proc. 25th IEEE Conf. Decision and Control*, Dec. 1986, pp. 1866-1870.

[13] F. Khorrami, "Analysis of multi-link flexible manipulators via asymptotic expansions," in *Proc. 28th IEEE Conf. Decision and Control*, Dec. 1989, pp. 2089-2094.

[14] P. V. Kokotovic, H. K. Kalil, and J. O'Reilly, *Singular perturbation methods in control: analysis and design*. London: Academic Press, 1986.

[15] P. V. Kokotovic, "Applications of singular perturbation techniques to control problems," *SIAM Review*, vol. 26, no. 4, pp.501-550, Oct. 1984.

[16] P. V. Kokotovic, R. E. O'Malley Jr., and P. Sannuti, "Singular perturbations and order reduction in control theory-An overview," *Automatica*, vol. 12, pp.123-132, 1976.

[17] F. L. Lewis, K. Liu, and A. Yeşildirek, "Neural net robot controller with guaranteed tracking performance," *IEEE Trans. Neural Networks,* vol.6, no. 3, pp. 703-715, May 1995.

[18] F. L. Lewis, S. Jagannathan, and A. Yeşildirek. *Neural Network Control of Robot Manipulators and Nonlinear Systems*. London: Taylor & Francis, to appear in 1997.

[19] F. L. Lewis, C. T. Abdallah, D. M. Dawson. *Control of robot manipulators*. New York: Macmillan, 1993.

[20] F. L. Lewis, K. Liu, and A. Yeşildirek, "Neural net robot controller: structure and stability proofs," in *Proc. 32nd IEEE Conf. Decision and Control,* Dec. 1993, pp. 2785-2791.

[21] J. Lin, and F. L. Lewis, "Dynamic equations of a manipulator with rigid and flexible links: derivation and symbolic computation," in *Proc. American Control Conf.*, Jun. 1993, pp. 2868-2872.

[22] S. H. Lin, S. Tosunoğlu, and D. Tesar, "Control of a six-degree-of-freedom flexible industrial manipulator," *IEEE Control Systems Magazine*, pp. 24-30, Apr. 1990.

[23] K. Liu, and F. L. Lewis, "Hybrid feedback linearization/fuzzy logic control of a flexible link manipulator," *J. Intelligent and Fuzzy Systems,* vol. 1, 1994.

[24] S. K. Madhavan and S. N. Singh, "Inverse trajectory control and zero dynamics sensitivity of an elastic manipulator," in *Proc. 1991 American Control Conference,* Jun.1991, pp. 1879-1884.

[25] W.T. Miller, R.S. Sutton, P.J. Werbos, ed., *Neural Networks for Control*. Cambridge, MA: MIT Press, 1991.

[26] K.S. Narendra, ``Adaptive Control Using Neural Networks," *Neural Networks for Control*, pp 115-142. ed. W.T. Miller, R.S. Sutton, P.J. Werbos, Cambridge, MA: MIT Press, 1991.

[27] K.S. Narendra and A.M. Annaswamy, ``A new adaptive law for robust adaptation without persistent excitation," *IEEE Trans. Automat. Control*, vol. AC-32, no. 2, pp. 134-145, 1987.

[28] P. J. Nathan and S. N. Singh, "Sliding mode control and elastic mode stabilization of a robotic arm with flexible links," *J. Dynam. Systems, Measurement and Control*, vol. 113, pp. 669-676, Dec. 1991.

[29] M.M. Polycarpou and P.A. Ioannou, ``Identification and control using neural network models: design and stability analysis," *Tech. Report 91-09-01*, Dept. Elect. Eng. Sys., Univ. S. Cal., Sept. 1991.

[30] G.A. Rovithakis and M.A. Christodoulou, ``Adaptive control of unknown plants using dynamical neural networks," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 24, no. 3, pp. 400-412, 1994.

[31] N. Sadegh, ``A perceptron network for functional identification and control of nonlinear systems," *IEEE Trans. Neural Networks*, vol. 4, no. 6, pp. 982-988, 1993.

[32] R.M. Sanner and J.-J.E. Slotine, ``Stable adaptive control and recursive identification using radial gaussian networks," in *Proc. IEEE Conf. Decision and Control*, Brighton, 1991.

[33] D. A. Schoenwald and Ü. Özgüner, "On combining slewing and vibration control in flexible manipulators via singular perturbations," in *Proc. 29th IEEE Conf. Decision and Control*, Dec. 1990, pp. 533-538.

[34] B. Siciliano and W. J. Book, "A singular perturbation approach to control of lightweight manipulators," *Int. J. Robotics Research*, vol.7, no. 4, pp.79-90, Aug. 1988.

[35] P. B. Usoro, R. Nadira, and S.S. Mahil, "A finite element/Lagrange approach to modeling lightweight flexible manipulators," *J. Dynam. Systems, Measurement and Control*, vol. 108, pp. 198-205, Sep. 1986.

[36] M. W. Vandegrift, and F. L. Lewis, and S. Zhu, "Flexible-link robot arm control by a feedback linearization/singular perturbation approach," *J. Robotic Systems*, vol. 11, no. 7, pp.591-603, 1994.

[37] D. Wang and M. Vidyagasar, "Transfer functions for a single flexible link," *Int. J. Robotics Research*, vol. 10, no. 5, pp. 540-549, Oct. 1991.

[38] D. Wang and M. Vidyagasar, "Control of a class of manipulators with a single flexible link - Part I: Feedback linearization," *J. Dynam. Systems, Measurement and Control*, vol. 113, pp. 655-661, Dec. 1991.

[39] D. Wang and M. Vidyagasar, "Control of a class of manipulators with a single flexible link - Part II: Observer-controller stabilization," *J. Dynam. Systems, Measurement and Control*, vol. 113, pp. 662-668, Dec. 1991.

[40] A. Yeşildirek, M. W. Vandegrift, and F. L. Lewis, "A neural network controller for flexible-link robots," in *IEEE Int. Symp. Intelligent Control*, Aug. 1994, pp. 63-68.